SAPIENZA
UNIVERSITÀ DI ROMA

LTL and Past LTL on Finite Traces for Planning and Declarative Process Mining

Faculty of Information Engineering, Informatics, and Statistics
Master of Science in Engineering in Computer Science

Candidate
Francesco Fuggitti
ID number 1735212

Thesis Advisor
Prof. Giuseppe De Giacomo

Academic Year 2017/2018

Thesis defended on 31st October 2018
in front of a Board of Examiners composed by:

Prof. Alberto Marchetti Spaccamela (chairman)
Prof. Silvia Bonomi
Prof. Francesco Delli Priscoli
Prof. Maurizio Lenzerini
Prof. Daniele Nardi
Prof. Leonardo Querzoni
Prof. Marco Schaerf

LTL and Past LTL on Finite Traces for Planning and Declarative Process Mining
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: fuggitti.1735212@studenti.uniroma1.it

Abstract

The application of LTL and Past LTL on Finite Traces is of particular interest for different fields in Artificial Intelligence (AI) and Computer Science (CS). In this thesis, we focus on the application of these two formal languages to Planning and Declarative Process Mining. For this purpose, the design and implementation of the LTL_f2DFA framework has been crucial to the application of LTL_f and PLTL in both research fields. In Planning, we direct our attention to FOND planning for LTL_f and PLTL goals showing a new encoding of those temporal goals in standard PDDL. We formally provide experimental evidence that it is possible to restrict the manner used by the plan to reach the goal specified in future and past modalities. With respect to Declarative Process Mining, we generalize the Janus approach for computing the interestingness of traces out of event logs, giving a new representation of the constraint formula allowing propositional formulas as activation condition.

Contents

Chapter 1

Introduction

1.1 Context

The literature on Artificial Intelligence (AI) and Computer Science (CS) has directed attention to Linear Temporal Logic (LTL) as the formal language for temporal specification of the sequence of actions of an agent or a system of agents (Fagin et al., 2004). Initially, LTL on infinite traces was formulated in Computer Science as a specification language for concurrent programs (Pnueli, 1977).

More specifically, the variant of LTL evaluated on *finite* traces (LTL_f) and its past counterpart Past LTL (PLTL), treated in this thesis, have been thoroughly investigated in De Giacomo and Vardi (2013); Lichtenstein et al. (1985). Concerning LTL_f, De Giacomo and Vardi (2013) conceptualise the encoding of LTL_f to First-Order Logic (FOL) by defining the translation function $fol(\varphi, x)$. Zhu et al. (2018) adopts this function, but modifying it with the appropriate built-in operators of MONA. MONA was formerly applied in Zhu et al. (2017) because of its performance. Regarding PLTL temporal specification, formerly studied in "The Glory of the Past" (Lichtenstein et al., 1985), Zhu et al. (2018) formulates the translation function $fol_p(\varphi, x)$ allowing the encoding of PLTL to FOL through the application of MONA operators. Nevertheless, there is no implementation of such a translation function with the MONA tool starting directly from a PLTL formula yet.

The above-mentioned formal languages, LTL_f and PLTL, are mechanism for specifying temporally extended goals in Planning and formula constraints in Business Process Modeling (BPM), more specifically in Declarative Process Mining. While Camacho et al. (2017) analyse non-deterministic planning for LTL on finite and infinite traces goals, academics have not investigated planning for PLTL goals. Regarding BPM, Cecconi et al. (2018), by employing the augmentation of LTL_f with past modalities, introduce a new approach to compute the ratio between the number of times a constraint formula is satisfied by a trace and the number of times the former is activated by the latter. However, such an approach conceptualises the activation condition of the constraint as a single task and it can only handle DECLARE constraints, under a practical perspective.

1.2 Objectives

This thesis sets specific objectives about the definition and application of LTL$_f$ and PLTL formalisms by referring to problems on cited research works.

Firstly, even though De Giacomo and Vardi (2013) formalized the theory behind the translation from an LTL$_f$ formula to FOL and Zhu et al. (2018) retrieved this approach customizing it for MONA, they have not yet implemented the translation functions both for LTL$_f$ and PLTL employing the MONA tool for the generation of the symbolic Deterministic Finite-state Automaton (DFA) with the sole purpose of directly translating formulas to DFA. In these terms, the implementation of our tool is critical for the optimization of the conversion processes of LTL$_f$ and PLTL formulas to DFA as shown in Zhu et al. (2017). Accordingly, the thesis will build a tool, named LTL$_f$2DFA, which will use MONA to directly convert both LTL$_f$ and PLTL formulas to their corresponding DFA.

Secondly, to the best of our knowledge, the literature on AI has not yet investigated planning for PLTL goals, i.e reaching a final state such that the history leading to such a state satisfies a PLTL formula. The investigation on planning for PLTL goals is of paramount importance in two respects. It generalizes the restriction of modes used by the plan to reach the goal and there is a computational advantage in using PLTL formulas directly when possible, since they can be reduced to DFA in single exponential time (vs. double-exponential time of LTL$_f$ formulas) (Chandra et al., 1981). While previous works (Camacho et al., 2017, 2018a,b) considered future goal specifications (i.e. using LTL$_f$ goals), the objective of this thesis is not only to allow past goal specifications (i.e. using PLTL goals), but also to provide a new formalization of those temporally extended goals in the Planning Domain Definition Language (PDDL). Such an objective will be attained through the result given by the LTL$_f$2DFA tool.

Finally, the thesis will propose a generalization of the Janus approach firstly introduced in Cecconi et al. (2018), under a theoretical and practical perspective. The Janus approach has two main drawbacks. In particular, the activation condition of a constraint formula could only be specified as a single task, since Cecconi et al. (2018) take into account only DECLARE constraints, thoroughly investigated in Pesic (2008). As a result, the constraint can be activated if, at a certain instant of the trace, one and only one task is executed. Secondly, the original Janus algorithm implementation does not include a tool to directly convert any LTL$_f$ and PLTL formulas into their related DFAs. The thesis will address such limitations by devising a generalization of the Janus approach, allowing any type of constraint and, then, employing the LTL$_f$2DFA tool to directly generate DFA of any LTL$_f$ and PLTL formulas.

1.3 Results

The thesis significantly contributes to the research areas of Formal Methods, Planning and Business Process Modeling.

In the first place, by directing attention to the interpretation of LTL and PLTL on *finite* traces, we designed and implemented a new tool, called LTL$_f$2DFA, that translates any LTL$_f$/PLTL formula to a Deterministic Finite-state Automaton (DFA). LTL$_f$2DFA is relevant in two respects. It is the first tool able to directly convert both LTL$_f$ and PLTL formulas to their corresponding DFA. Secondly, it adopts the MONA tool for the generation of automata. Accordingly, we have advanced researches in Zhu et al. (2017, 2018), by delivering the LTL$_f$2DFA software package. Moreover, the LTL$_f$2DFA is also available on http://ltlf2dfa.diag.uniroma1.it.

Concerning Planning, we have explored how LTL$_f$ and PLTL can be used for expressing extended temporal goals in *fully observable non-deterministic* (FOND) planning problems. In these terms, we have proposed a new approach in compiling temporally extended goals together with the original planning domain, specified in PDDL, which is suitable for input to standard (FOND) planners (e.g. FOND-SAT planner in Geffner and Geffner (2018)). The encoding of those temporal goals, directly in the PDDL domain and problem, relies on the result given by LTL$_f$2DFA. More specifically, we have encoded DFA transitions as a new PDDL operator and modified the goal accordingly. The absolute novelty is that, given the new LTL$_f$2DFA tool, it is now possible to express temporal extended goals not only in LTL$_f$, but also in PLTL (i.e. with past modalities), unlike former researches in this area of application (Camacho et al., 2017, 2018a,b).

Regarding BPM, LTL$_f$2DFA enables the extension and generalization of the Janus approach in declarative process mining for computing the *interestingness degree* of traces in event logs. From a theoretical perspective, we have generalized the Janus approach by giving a new representation of the constraint formula allowing propositional formulas as activation condition, rather than a single task as in Cecconi et al. (2018). From a practical perspective, we have implemented this modified approach by exploiting the power of the LTL$_f$2DFA tool. In such a scenario, LTL$_f$2DFA has allowed to generate, at execution time, DFAs for any type of formula, overcoming the original limitation of the Janus approach to DECLARE constraints.

1.4 Structure of the Thesis

The thesis is structured as follows:

- In Chapter 2, we will illustrate the theoretical framework, consisting of LTL, LTL$_f$ and PLTL formalisms, underlying the thesis. These formal languages will be described focusing the attention on their syntax, semantics and interesting properties. Besides, we will be talking about the theory behind the translation procedure of LTL$_f$ and PLTL formulas to DFAs. Finally, we will present the MONA tool explaining in details the encoding process starting from an LTL$_f$/PLTL formula to a MONA program passing through a FOL translation.

- In Chapter 3, we will present the LTL$_f$2DFA Python package. We will also describe the structure of the package, discussing in detail its implementation highlighting all the main features and, finally, seeing how it performs in time relatively to the FLLOAT Python package.

- In Chapter 4, we will face the problem of FOND Planning for LTL$_f$/PLTL goals. In particular, we will propose a new solution, called FOND4LTL$_f$/PLTL, that essentially reduces the problem to a "classical" FOND planning problem. This will be possible thanks to our LTL$_f$2DFA Python tool which will be employed for the encoding of temporally extended goals into standard PDDL. Then, we will also described in details the FOND4LTL$_f$/PLTL implementation, highlighting all its main features. Finally, we will see examples of execution results.

- In Chapter 5, we will present how the LTL$_f$2DFA Python package has been well employed in the field of Business Process Management. In particular, we will explore the Janus approach to declarative process mining enhancing its peculiarities and, at the same time, giving our substantial contributions in generalizing the approach itself. After that, we will describe the implementation of the JANUS algorithm, modified accordingly, highlighting all its main features. Finally, we will see examples of execution results.

- In Chapter 6, the thesis will conclude summarizing its main achievements and discussing possible future works.

Chapter 2

PLTL and LTL_f

This chapter will deal with the theoretical framework on which all topics present in the thesis are based. Initially, we will introduce the widely known Linear-Time Temporal Logic (LTL) and the Past Linear Time Temporal Logic (PLTL), focusing on their syntax and semantic. Secondly, we will talk about the concept of *Finite Trace* in these formal languages and how it changes them. Specifically, we will describe the Linear Time Temporal Logic over Finite Traces (LTL_f). Then, we will illustrate the theory behind the transformation of an LTL_f or PLTL formula to a Deterministic Finite State Automaton (DFA). Finally, we will describe the translation of an LTL_f or PLTL formula to the classic First-Order Logic formalism (FOL) and the translation of a FOL formula into a program that the MONA, a tool that translates formulas into a DFA, can manage. Some examples will be provided, but we will suppose the reader to be confident with classical logic and automata theory.

2.1 Linear Temporal Logic (LTL)

Temporal Logic formalisms are a set of formal languages designed for representing temporal information and reasoning about time within a logical framework (Goranko and Galton, 2015). Indeed, these logics are used when propositions have their truth value dependent on time.

In this scenario, we find the *Linear Temporal Logic* (LTL) which is a a very well known modal temporal logic with modalities referring to time. It was originally proposed in (Pnueli, 1977) as a specification language for concurrent programs. Consequently, LTL has been extensively used in Artificial Intelligence and Computer Science. For instance, it has been employed in planning, reasoning about actions, declarative process mining and verification of software/hardware systems.

2.1.1 Syntax

Given a set of propositional symbols $\mathcal{P}$, a valid LTL formula φ is defined as follows:

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where $a \in \mathcal{P}$. The unary operator $\bigcirc$(*next-time*) and the binary operator $\mathcal{U}$ (*until*) are temporal operators and we use $\top$ and $\bot$ to denote *true* and *false* respectively. Moreover, all classical logic operators $\vee, \Rightarrow, \Leftrightarrow, true$ and $false$ can be used. Intuitively, $\bigcirc\varphi$ says that φ is true at the *next* instant, $\varphi_1 \mathcal{U} \varphi_2$ says that at some future instant, φ_2 will hold and *until* that point φ_1 holds. We also define common abbreviations for some specific temporal formulas: *eventually* as $\Diamond\varphi \doteq true \mathcal{U} \varphi$, *always* as $\Box\varphi \doteq \neg\Diamond\neg\varphi$, *weak-next* as $\bullet\varphi \doteq \neg\bigcirc\neg\varphi$ and *release* as $\varphi_1 \mathcal{R} \varphi_2 \doteq \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$.

LTL allows to express a lot of interesting properties defined over time. In the Example 2.1 we show some of them.

Example 2.1. Interesting LTL patterns:

- *Safety*: $\Box\varphi$, which means *it is always true that property in φ will happen* or *φ will hold forever.* For instance, $\Box\neg(reactorTemp > 1000)$ (the temperature of the reactor must never exceed 1000).

- *Liveness*: $\Diamond\varphi$, which means *sooner or later φ will hold* or *something good will eventually happen.* For instance, $\Diamond rich$ (eventually I will become rich).

- *Response*: $\Box\Diamond\varphi$ which means *for every point in time, there is a point later where φ holds.*

- *Persistence*: $\Diamond\Box\varphi$, which means *there exists a point in the future such that from then on φ always holds.*

- *Strong fairness*: $\Box\Diamond\varphi_1 \Rightarrow \Box\Diamond\varphi_2$, *if something is attempted/requested infinitely often, then it will be successful/allocated infinitely often.* For instance, $\Box\Diamond ready \Rightarrow \Box\Diamond run$ (if a process is in ready state infinitely often, then it will be selected by the scheduler infinitely often).

2.1.2 Semantics

The semantics of the main operators of LTL over *infinite traces* are expressed as an ω-word over the alphabet $2^{\mathcal{P}}$. We give the following definitions:

Definition 2.1. Given an infinite trace π, we inductively define when an LTL formula φ is *true* at an instant i, in symbols $\pi, i \models \varphi$, as follows:

$$\pi, i \models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$

$\pi, i \models \circ\varphi$ iff $\pi, i+1 \models \varphi$

$\pi, i \models \varphi_1 \mathcal{U} \varphi_2$ iff $\exists j.(j \geq i) \wedge \pi, j \models \varphi_2 \wedge \forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1$

Definition 2.2. An LTL formula φ is *true* in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is *satisfiable* if it is true in some π and is *valid* if it is true in every π. A formula φ_1 *logically implies* another formula φ_2, in symbols $\varphi_1 \models \varphi_2$ iff $\forall \pi, \pi \models \varphi_1 \Rightarrow \pi \models \varphi_2$.

Notice that satisfiability, validity and logical implication are all mutually reducible one to each other.

Example 2.2. Validity and logical implication as satisfiability

- φ is valid iff $\neg\varphi$ is unsatisfiable.
- $\varphi_1 \models \varphi_2$ iff $\varphi_1 \wedge \neg\varphi_2$ is unsatisfiable.

2.1.3 Results

About LTL complexity, we can state the following fundamental theorem:

Theorem 2.1. *(Sistla and Clarke, 1985) Satisfiability, validity, and logical implication for* LTL *formulas are* PSPACE*-complete.*

2.2 Linear Temporal Logic on Finite Traces (LTL$_f$)

Linear Temporal Logic on Finite Traces (LTL$_f$) is the variant of LTL described in Section 2.1 interpreted over *finite traces* (De Giacomo and Vardi, 2013). Although it seems a little difference, in some cases, the interpretation of a formula over finite traces completely changes its meaning with respect to the one over inifinite traces.

2.2.1 Syntax

The syntax of LTL$_f$ is exactly the same of LTL. Indeed, LTL$_f$ formulas are built from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\circ$(*next-time*) and the binary operator $\mathcal{U}$ (*until*). Formulas can be defined as follows:

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \circ\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where $a \in \mathcal{P}$. All usual logical operators such as $\vee, \Rightarrow, \Leftrightarrow, true$ and $false$ are also used. Similarly to LTL, we can define the following common abbreviations for temporal operators:

$$\Diamond\varphi \doteq true \mathcal{U} \varphi \tag{2.1}$$

$$\Box\varphi \doteq \neg\Diamond\neg\varphi \tag{2.2}$$

$$\bullet\varphi \doteq \neg\circ\neg\varphi \tag{2.3}$$

$$\varphi_1 \,\mathcal{R}\, \varphi_2 \doteq \neg(\neg\varphi_1 \,\mathcal{U}\, \neg\varphi_2) \tag{2.4}$$

$$\mathit{Last} \doteq \bullet\mathit{false} \tag{2.5}$$

$$\mathit{End} \doteq \Box\mathit{false} \tag{2.6}$$

Compared with LTL, in LTL$_f$ there have been defined also 2.5 and 2.6 which denotes the last instance of the trace and that the trace is ended, respectively. As we have seen in Example 2.1 with LTL, now we will see in Example 2.3 how properties expressed in LTL$_f$ have changed their meaning with the interpretation over finite traces.

Example 2.3. Interesting LTL$_f$ patterns:

- *Safety*: $\Box\varphi$, which now means always *till the end of the trace* φ holds.
- *Liveness*: $\Diamond\varphi$, which now means eventually *before the end of the trace* φ holds.
- *Responce*: $\Box\Diamond\varphi$, which means for any point in the trace there exist a point later in the trace where φ holds. This property, interpreted over finite traces, can be seen also as $\Diamond(\mathit{Last} \wedge \varphi)$ because $\Box\Diamond\varphi$ implies that the *last point in the trace satisfies* φ.
- *Persistance*: $\Diamond\Box\varphi$ means that there is a point in the trace such that from then on until the end of the trace φ holds. Also here the meaning can me seen as $\Diamond(\mathit{Last} \wedge \varphi)$ since $\Diamond\Box\varphi$ implies that at the last point of the trace $\Box\varphi$, and so φ, holds.

In other words, no direct nesting of *eventually* and *always* connectives is meaningful in LTL$_f$. However, indirect nesting of *eventually* and *always* connectives can still produce meaningful and interesting properties. One example could be $\Box(\psi \Rightarrow \Diamond\varphi)$, which stands for *always, before the end of the trace, if* ψ *holds then* φ *will eventually hold.*

2.2.2 Semantics

The semantics of LTL$_f$ is given as LT$_f$-interpretations, namely interpretations over a *finite traces* denoting a finite sequence of consecutive instants of time. Formally, LT$_f$-interpretations are expressed as finite words π over the alphabet $2^{\mathcal{P}}$, i.e. as alphabet we have all the possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We use the following notation. We denote the *length* of a trace π as $\mathit{length}(\pi)$. We denote the *positions*, i.e. instants, on the trace as $\pi(i)$ with $0 \leq i \leq \mathit{last}$ where $\mathit{last} = \mathit{length}(\pi) - 1$ is the last element of the trace. We denote by $\pi(i, j)$, the *segment* (i.e., the subword) of π, the trace $\pi' = \langle \pi(i), \pi(i+1), \ldots, \pi(j) \rangle$, with $0 \leq i \leq j \leq \mathit{last}$. We now give the following definitions:

Definition 2.3. Given an LT_f-interpretation π, we define when an LTL_f formula φ is *true* at position i (for $0 \leq i \leq last$), in symbols $\pi, i \models \varphi$, inductively as follows:

$$\pi, i \models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i)$$

$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$

$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$

$$\pi, i \models \circ\varphi \text{ iff } i < last \wedge \pi, i+1 \models \varphi \tag{2.7}$$

$$\pi, i \models \varphi_1 \,\mathcal{U}\, \varphi_2 \text{ iff } \exists j.(i \leq j \leq last) \wedge \pi, j \models \varphi_2 \wedge \forall k.(i \leq k < j) \Rightarrow \pi, k \models \varphi_1 \tag{2.8}$$

The Definition 2.3 is exactly the same Definiton 2.1 seen for LTL except for 2.7 and 2.8 in which the only difference lies on the intervals bounded by the last element of the trace.

Definition 2.4. An LTL_f formula is *true* in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula φ is *satisfiable* if it is true in some LT_f-interpretation, and is *valid* if it is true in every LT_f-interpretation. A formula φ_1 *logically implies* another formula φ_2, in symbols $\varphi_1 \models \varphi_2$ iff for every LT_f-interpretation π we have that $\pi \models \varphi_1$ implies $\pi \models \varphi_2$.

2.2.3 Results

About LTL_f complexity, we can state the following theorem:

Theorem 2.2. *(De Giacomo and Vardi, 2013) Satisfiability, validity and logical implication for* LTL_f *formulas are* PSPACE*-complete.*

About LTL_f expressiveness, we have that:

Theorem 2.3. *(De Giacomo and Vardi, 2013; Gabbay et al., 1997)* LTL_f *has exactly the same expressive power of* FOL *over finite ordered sequences.*

2.3 Past Linear Temporal Logic (PLTL)

So far we have seen LTL and LTL_f languages, over infinite and finite traces respectively, that look into the future events. On the contrary, now we describe the so called *Past Linear Temporal Logic* (PLTL) which is the counterpart of the LTL and LTL_f because it uses temporal modalities for referring to past events, instead of future ones.

2.3.1 Syntax

The syntax of PLTL is exactly the same of the one seen in Section 2.1.1 for LTL and in Section 2.2.1 for LTL_f except for past temporal operators that are the inverse of the future ones. As stated before, PLTL formulas are built on top from a set $\mathcal{P}$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\ominus$(*previous-time*) and the binaty operator $\mathcal{S}$ (*since*). Formulas can be defined as follows:

$$\varphi \quad ::= \quad a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \ominus\varphi \mid \varphi_1 \,\mathcal{S}\, \varphi_2$$

where $a \in \mathcal{P}$. All usual logical operators such as $\vee, \Rightarrow, \Leftrightarrow, true$ and $false$ can be derived. Similarly to LTL and LTL$_f$, we define the following common abbreviations for temporal operator:

$$\Diamond\!\!\!-\varphi \doteq true\,\mathcal{S}\,\varphi \tag{2.9}$$

$$\boxminus\varphi \doteq \neg\Diamond\!\!\!-\neg\varphi \tag{2.10}$$

In particular, $\Diamond\!\!\!-\varphi$ in 2.9 is called *once* while $\boxminus\varphi$ in 2.10 is known as *historically.* Furthermore, both temporal operators *previous-time*, *since* and the two common abbreviations *once*, *historically* just defined above could be seen also as the inverse operators of future operators in LTL/LTL$_f$:

$$\ominus\varphi \equiv \bigcirc^{-1}\varphi$$

$$\varphi_1 \,\mathcal{S}\, \varphi_2 \equiv \varphi_1 \mathcal{U}^{-1} \varphi_2$$

$$\Diamond\!\!\!-\varphi \equiv \Diamond^{-1}\varphi$$

$$\boxminus\varphi \equiv \Box^{-1}\varphi$$

2.3.2 Semantics

As we did previously with LTL and then with LTL$_f$, here we define a semantics to PLTL. The first important thing to notice is that a PLTL formula could be only interpreted over *finite* traces. This is due to the fact that, no matter how long the trace is, there must be a starting point in the past. Formally, a trace π is a word over the alphabet $2^{\mathcal{P}}$ and as alphabet we have all possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We can now give the following definitions:

Definition 2.5. Given a trace π, we inductively define when a PLTL formula φ is *true* at time i, in symbols $\pi, i \models \varphi$, as follows:

$$\pi, i \models a, \text{ for } a \in \mathcal{P} \text{ iff } a \in \pi(i)$$
$$\pi, i \models \neg\varphi \text{ iff } \pi, i \not\models \varphi$$
$$\pi, i \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi, i \models \varphi_1 \wedge \pi, i \models \varphi_2$$
$$\pi, i \models \ominus\varphi \text{ iff } i > 0 \wedge \pi, i-1 \models \varphi$$
$$\pi, i \models \varphi_1 \,\mathcal{S}\, \varphi_2 \text{ iff } \exists j.(j \leq i) \wedge \pi, j \models \varphi_2 \wedge \forall k.(j < k \leq i) \Rightarrow \pi, k \models \varphi_1$$

The Definition 2.5 is quite similar to Definitions 2.1 and 2.3. The only difference lies on the position in time of instances, indeed, in this case, we go backward.

2.3.3 Results

About PLTL complexity, we can state the following theorem:

Theorem 2.4. *Satisfiability, validity and logical implication for* PLTL *formulas are* PSPACE-*complete.*

2.3.4 Expressiveness

About expressiveness of PLTL, we can state the following theorem:

Theorem 2.5. PLTL *has exactly the same expressive power of* LTL$_f$.

Here, it is worth to say that the LTL formalism augmented with past temporal operators present in PLTL can be exponentially more succinct than LTL (with only future operators) (Markey, 2003). Indeed, having at the same time past and future temporal operators is really useful because, in general, expressions given in natural language use references to events occurred in the past. We give an example in the following.

Example 2.4. Succinctness of LTL with Past:

$$\Box(grant \Rightarrow \Diamond\!\!\!\!-\, request) \tag{2.11}$$

$$\neg((\neg request)\,\mathcal{U}(grant \wedge \neg request)) \tag{2.12}$$

Both formulas mean *every grant is preceeded by a request.* The former (2.11) is in LTL with past modalities whereas the latter (2.12) is pure LTL. It is pretty evident that the 2.11 is less intricate than the one in 2.12.

Finally, this property of LTL augmented with past temporal operators is interesting, however it is out of the scope of this thesis.

2.4 LTL$_f$ and PLTL Translation to Automata

Given an LTL$_f$/PLTL formula φ, we can build a deterministic finite state automaton (DFA) (Rabin and Scott, 1959) $\mathcal{A}_\varphi$ that accepts the same finite traces that makes φ true. To accomplish this, we proceed in two steps: first, we translate LTL$_f$ and PLTL formulas into an (NFA) (De Giacomo and Vardi, 2015) following a simple direct algorithm; secondly, the obtained NFA can be converted into a DFA following the standard *determinization* procedure.

Now, we recall definitions of NFA and DFA:

Definition 2.6. An NFA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where:

- Σ is the input alphabet;
- Q is the finite set of states;
- $q_0 \in Q$ is the initial state;

- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation;
- $F \subseteq Q$ is the set of final states;

Definition 2.7. A DFA is a NFA where δ is a function $\delta : Q \times \Sigma \to Q$

To denote the set of all traces over Σ accepted by $\mathcal{A}$ we will use $\mathcal{L}(A)$ henceforth. In the next subsections, we will provide some definitions and we will illustrate the algorithm for the translation.

2.4.1 ∂ function for LTL$_f$

To build the NFA, we need to define an auxiliary function ∂.

Definition 2.8. The *delta function ∂ for LTL$_f$ formulas* is a function that takes as input an (implicitly quoted) LTL$_f$ formula φ (in negation normal form (NNF)[1]) and a propositional interpretation Π for $\mathcal{P}$ (including *Last*), and returns a positive boolean formula whose atoms are (implicitly quoted) φ subformulas. It is defined as follows:

$$\begin{aligned}
\partial(A, \Pi) &= \begin{cases} \mathit{true} & \text{if } A \in \Pi \\ \mathit{false} & \text{if } A \notin \Pi \end{cases} \\
\partial(\neg A, \Pi) &= \begin{cases} \mathit{false} & \text{if } A \in \Pi \\ \mathit{true} & \text{if } A \notin \Pi \end{cases} \\
\partial(\varphi_1 \wedge \varphi_2, \Pi) &= \partial(\varphi_1, \Pi) \wedge \partial(\varphi_2, \Pi) \\
\partial(\varphi_1 \vee \varphi_2, \Pi) &= \partial(\varphi_1, \Pi) \vee \partial(\varphi_2, \Pi) \\
\partial(\circ\varphi, \Pi) &= \varphi \wedge \neg \mathit{End} \equiv \varphi \wedge \Diamond \mathit{true} \\
\partial(\varphi_1 \mathcal{U} \varphi_2, \Pi) &= \partial(\varphi_2, \Pi) \vee (\partial(\varphi_1, \Pi) \wedge \partial(\circ(\varphi_1 \mathcal{U} \varphi_2), \Pi)) \\
\partial(\bullet\varphi, \Pi) &= \varphi \vee \mathit{End} \equiv \varphi \vee \Box \mathit{false} \\
\partial(\varphi_1 \mathcal{R} \varphi_2, \Pi) &= \partial(\varphi_2, \Pi) \wedge (\partial(\varphi_1, \Pi) \vee \partial(\bullet(\varphi_1 \mathcal{R} \varphi_2), \Pi))
\end{aligned} \tag{2.13}$$

where *End* is defined as Equation 2.6. As a consequence of Definition 2.8 and from Equation 2.1 and 2.2, we can deduce that

$$\begin{aligned}
\partial(\Diamond\varphi, \Pi) &= \partial(\varphi, \Pi) \vee \partial(\circ\Diamond\varphi, \Pi) \\
\partial(\Box\varphi, \Pi) &= \partial(\varphi, \Pi) \wedge \partial(\bullet\Box\varphi, \Pi)
\end{aligned}$$

[1] A formula is in *negation normal form* if negation ($\neg$) occurs only in front of atoms.

Moreover, we define $\partial(\varphi, \epsilon)$ which is inductively defined as Equation 2.13, except for the following cases:

$$\begin{aligned}
\partial(A, \epsilon) &= \mathit{false} \\
\partial(\neg A, \epsilon) &= \mathit{false} \\
\partial(\circ\varphi, \epsilon) &= \mathit{false} \\
\partial(\bullet\varphi, \epsilon) &= \mathit{true} \\
\partial(\varphi_1 \,\mathcal{U}\, \varphi_2, \epsilon) &= \mathit{false} \\
\partial(\varphi_1 \,\mathcal{R}\, \varphi_2, \epsilon) &= \mathit{true}
\end{aligned} \tag{2.14}$$

Notice that $\partial(\varphi, \epsilon)$ is always either *true* or *false*. Here, it is worth to observe that from Equation 2.14 we can say $\partial(\Diamond\varphi, \epsilon) = \mathit{false}$ and $\partial(\Box\varphi, \epsilon) = \mathit{true}$.

2.4.2 ∂ function for PLTL

Since, from Theorem 2.5, PLTL has exactly the same expressive power of LTL$_f$, the corresponding ∂ function for PLTL is equivalent to the one seen in Section 2.4.1, but using past temporal operators. At this point, we should formally define how to get the LTL$_f$ formula corresponding to a PLTL formula. We give the following definition and theorem.

Definition 2.9. Given a PLTL formula φ, the inverted LTL$_f$ formula φ^R of φ is obtained by replacing all temporal operators in φ with the corresponding future operators.

Theorem 2.6. *Let $\mathcal{L}(\varphi)$ be the language of a PLTL formula φ and $\mathcal{L}^R(\varphi)$ be its reversed language, then $\mathcal{L}(\varphi) = \mathcal{L}^R(\varphi^R)$*

The Theorem 2.6 shows that the reversed LTL$_f$ formula φ^R accepts exactly the reversed language satisfying the PLTL formula φ.

Example 2.5. Given a PLTL formula $\varphi = \boxminus(a \Rightarrow \ominus b)$, its reversed formula φ^R is obtained by replacing all past temporal operators $\boxminus$ and $\ominus$ with their dual in the future, respectively $\Box$ and $\circ$. So, we have $\varphi^R = \Box(a \Rightarrow \circ b)$.

2.4.3 The LTL$_f$2NFA algorithm

The LTL$_f$2NFA algorithm takes as input an LTL$_f$/PLTL formula φ and outputs an NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F\rangle$ that accepts exactly the traces satisfying φ. The algorithm, presented in the following, is a variant of the algorithm in (De Giacomo and Vardi, 2015).

Moreover, its correctness relies on the fact that every LTL$_f$/PLTL formula φ can be associated to a polynomial *alternating automaton on words* (AFW) accepting exactly the traces that make φ true and that every AFW can be transformed into an NFA (De Giacomo and Vardi, 2013). Furthermore, the algorithm requires that φ is in *negation normal form* (NNF), which can be done in linear time.

The function ∂ used in lines 5, 12 and 15 is the one defined in Section 2.4.1.

Algorithm 2.1. LTL$_f$2NFA: from LTL$_f$/PLTL formula φ to NFA $\mathcal{A}_\varphi$

1: **input** LTL$_f$/PLTL formula φ
2: **output** NFA $\mathcal{A}_\varphi = \langle 2^{\mathcal{P}}, Q, q_0, \delta, F\rangle$
3: $q_0 \leftarrow \{\varphi\}$
4: $F \leftarrow \{\emptyset\}$
5: **if** $(\partial(\varphi, \epsilon) = \mathit{true})$ **then**
6:     $F \leftarrow F \cup \{q_0\}$
7: **end if**
8: $Q \leftarrow \{q_0, \emptyset\}$
9: $\delta \leftarrow \emptyset$
10: **while** (Q or δ change) **do**
11:     **for** $(q \in Q)$ **do**
12:         **if** $(q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi))$ **then**
13:             $Q \leftarrow Q \cup \{q'\}$
14:             $\delta \leftarrow \delta \cup \{(q, \Pi, q')\}$
15:             **if** $(\bigwedge_{(\psi \in q')} \partial(\psi, \epsilon) = \mathit{true})$ **then**
16:                 $F \leftarrow F \cup \{q'\}$
17:             **end if**
18:         **end if**
19:     **end for**
20: **end while**

How LTL$_f$2NFA works

The LTL$_f$2NFA algorithm proceeds in a forward fashion. Indeed, the algorithm visits every state already seen q, checks all the possible transitions from that state and collects the results until states and transitions do not change. Consequently, it computes the next state q', the new transition (q, Π, q') and whether q' is a final state or not. Intuitively, the delta function ∂ emulates the semantic behaviour of every LTL$_f$/PLTL subformula after seeing Π.

States of $\mathcal{A}_\varphi$ are sets of atoms (each atom is a quoted φ subformula) to be interpreted as conjunctions. The empty conjunction $\emptyset$ stands for *true*. Moreover, q' is a set of quoted subformulas of φ denoting a minimal interpretation such that $q' \models \bigwedge_{(\psi \in q)} \partial(\psi, \Pi)$ (notice that we have also $(\emptyset, p, \emptyset) \in \delta$ for every $p \in 2^{\mathcal{P}}$).

The following result holds:

Theorem 2.7 (De Giacomo and Vardi (2015)). *Algorithm* LTL$_f$2NFA *is correct, i.e., for every finite trace* $\pi : \pi \models \varphi$ *iff* $\pi \in \mathcal{L}(\mathcal{A}_\varphi)$. *Moreover, it terminates in at most an exponential number of steps, and generates a set of states* S *whose size is at most exponential in the size of the formula* φ.

In order to obtain a DFA, the NFA $\mathcal{A}_\varphi$ can be determinized in exponential time (Rabin and Scott, 1959). Hence, an LTL$_f$ formula can be transformed into the corresponding DFA in 2EXPTIME. On the other hand, for PLTL formulas there is a computational advantage, since they can be reduced to DFA in single exponential time (vs. double-exponential time of LTL$_f$ formulas). This is due to the possibility of obtaining the DFA for the reverse of an alternating automaton in EXPTIME (Chandra et al., 1981).

2.5 LTL$_f$/PLTL to FOL Encoding and MONA

In this section, we will illustrate how to translate an LTL$_f$ and a PLTL formula into *first-order logic* (FOL) over finite linear ordered sequences[2] (De Giacomo and Vardi, 2013; Zhu et al., 2018). Then, we will present the MONA tool with its syntax and we will explain the translation procedure from a FOL encoding to the MONA enconding.

2.5.1 LTL$_f$-to-FOL Encoding

In the following we deal with a first-order language augmented with monadic predicates *succ*, $<$ and $=$ plus two constants 0 and *last*. Afterwards, we focus our attention to *finite linear ordered* FOL *interpretations* under the form of $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, where the domain is $\Delta^I = \{0, \ldots, n\}$ with $n \in \mathbb{N}$, and the interpretation function $\cdot^{\mathcal{I}}$ interprets binary predicates and constants as follows:

$$
\begin{aligned}
succ^{\mathcal{I}} &= \{(i, i+1) \mid i \in \{0, \ldots, n-1\}\} \\
<^{\mathcal{I}} &= \{(i, j) \mid i, j \in \{0, \ldots, n\} \wedge i < j\} \\
=^{\mathcal{I}} &= \{(i, i) \mid i \in \{0, \ldots, n\}\} \\
0^{\mathcal{I}} &= 0 \\
last^{\mathcal{I}} &= n
\end{aligned}
\tag{2.15}
$$

Actually, all these operators can be derived from $<$ as follows:

$$
\begin{aligned}
succ(x, y) &\doteq x < y \wedge \neg\exists z. x < z < y \\
x = y &\doteq \forall z. x < z \equiv y < z \\
0 &\doteq x \mid \neg\exists y. succ(y, x) \\
last &\doteq x \mid \neg\exists y. succ(x, y)
\end{aligned}
$$

[2]More precisely *monadic first-order logic on finite linearly ordered domains*, sometimes denoted as FO[<].

Although there could be possible differences in notation, the relation between LT$_f$-intepretations and finite linear ordered FOL interpretations is isomorphic. Indeed, given an LT$_f$-interpretation π we can define the corresponding FOL interpretation $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$ as follows: $\Delta^I = \{0, \ldots, last\}$, with $last = length(\pi) - 1$, with the predefined predicates and constants interpretation and, for each $a \in \mathcal{P}$ its interpretation is $a^{\mathcal{I}} = \{i \mid a \in \pi(i)\}$. On the contrary, given a finite linear ordered FOL interpretation $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, with $\Delta^I = \{0, \ldots, n\}$, we determine the corresponding LT$_f$-interpretation π as follows: $length(\pi) = n + 1$, for each instant $0 \leq i \leq last$ (with $last = n$), we obtain $\pi(i) = \{a \mid i \in a^{\mathcal{I}}\}$.

At this moment, we can define the translation function $fol(\varphi, x)$ in the following way.

Definition 2.10. Given an LTL$_f$ formula φ and a variable x, the translation function $fol(\varphi, x)$, inductively defined on the LTL$_f$ formula's structure, returns the corresponding FOL formula open in x:

$$fol(a, x) = a(x)$$

$$fol(\neg\varphi, x) = \neg fol(\varphi, x)$$

$$fol(\varphi_1 \wedge \varphi_2, x) = fol(\varphi_1, x) \wedge fol(\varphi_2, x)$$

$$fol(\varphi_1 \vee \varphi_2, x) = fol(\varphi_1, x) \vee fol(\varphi_2, x)$$

$$fol(\circ\varphi, x) = \exists y.succ(x, y) \wedge fol(\varphi, y)$$

$$fol(\bullet\varphi, x) = x = last \vee \exists y.succ(x, y) \wedge fol(\varphi, y)$$

$$fol(\varphi_1 \,\mathcal{U}\, \varphi_2, x) = \exists y.x \leq y \leq last \wedge fol(\varphi_2, y) \wedge \forall z.x \leq z < y \Rightarrow fol(\varphi_1, z)$$

$$fol(\varphi_1 \,\mathcal{R}\, \varphi_2, x) = \exists y.x \leq y \leq last \wedge fol(\varphi_1, y) \wedge \forall z.x \leq z < y \Rightarrow fol(\varphi_2, z) \vee$$

$$\forall z.x \leq z < last \Rightarrow fol(\varphi_2, z)$$

The following Theorem ensures that a finite trace ρ satisfies an LTL$_f$ formula φ iff the corresponding finite linear ordered FOL interpretation $\mathcal{I}$ of ρ models $fol(\varphi, 0)$.

Theorem 2.8. *(De Giacomo and Vardi, 2013) Given an* LT$_f$*-interpretation* π *and a corresponding finite linear ordered* FOL *interpretation* $\mathcal{I}$*, we have:*

$$\pi, i \models \varphi \textit{ iff } I, [x/i] \models fol(\varphi, x)$$

where $[x/i]$ *stands for a variable assignments that assigns the value* i *to the free variable* x *of* $fol(\varphi, x)$*.*

In general, recalling the Definition 2.4, a formula φ is *true* in a trace π ($\pi \models \varphi$) if $\pi, 0 \models \varphi$. Hence, we should evaluate our translation function $fol(\varphi, x)$ in 0 (i.e. computing $fol(\varphi, 0)$). Finally, since also the converse reduction of Theorem 2.8 holds, we can state the following Theorem:

Theorem 2.9. *(Gabbay et al., 1980)* LTL$_f$ *has exactly the same expressive power of* FOL*.*

2.5.2 PLTL-to-FOL Encoding

As we have previously seen for LTL$_f$, in the current section we describe the translation function for a PLTL formula. Here, we also have a first-order language augmented with monadic predicates *prev*, $<$ and $=$ plus two constants 0 and *last*. Then, we have our *finite linear ordered* FOL *interpretations* under the form of $\mathcal{I} = (\Delta^I, \cdot^{\mathcal{I}})$, where the domain is $\Delta^I = \{0, \ldots, n\}$ with $n \in \mathbb{N}$, and the interpretation function $\cdot^{\mathcal{I}}$ interprets the same binary predicates defined as 2.15 except that here we change *succ* with *prev* defined as follows:

$$prev^{\mathcal{I}} = \{(i, i-1) \mid i \in \{1, \ldots, n\}\} \tag{2.16}$$

We can derive these operators from $<$ as well:

$$\begin{aligned} prev(x,y) &\doteq y < x \wedge \neg\exists z.y < z < x \\ x = y &\doteq \forall z.x < z \equiv y < z \\ 0 &\doteq x \mid \neg\exists y.prev(x,y) \\ last &\doteq x \mid \neg\exists y.prev(y,x) \end{aligned}$$

In the exactly same way done before, we can give the definition of the translation function $fol_p(\varphi, x)$:

Definition 2.11. Given a PLTL formula φ and a variable x, the translation function $fol_p(\varphi, x)$, inductively defined on the PLTL formula's structure, returns the corresponding FOL formula open in x:

$$\begin{aligned} &fol_p(a, x) = a(x) \\ &fol_p(\neg\varphi, x) = \neg fol_p(\varphi, x) \\ &fol_p(\varphi_1 \wedge \varphi_2, x) = fol_p(\varphi_1, x) \wedge fol_p(\varphi_2, x) \\ &fol_p(\varphi_1 \vee \varphi_2, x) = fol_p(\varphi_1, x) \vee fol_p(\varphi_2, x) \\ &fol_p(\ominus\varphi, x) = \exists y.prev(x,y) \wedge y \geq 0 \wedge fol_p(\varphi, y) \\ &fol_p(\varphi_1 \mathcal{S} \varphi_2, x) = \exists y.0 \leq y \leq x \wedge fol_p(\varphi_2, y) \wedge \forall z.y < z \leq x \Rightarrow fol_p(\varphi_1, z) \end{aligned}$$

Consider a finite trace ρ, the corresponding FOL interpretation $\mathcal{I}$ is defined as in Section 2.5.1. The following Theorem ensures that a finite trace ρ satisfies an PLTL formula φ iff the corresponding finite linear ordered FOL interpretation $\mathcal{I}$ of ρ models $fol_p(\varphi, last)$.

Theorem 2.10. *(Kamp, 1968) Given a* PLTL *formula φ, a finite trace ρ, and the corresponding interpretation $\mathcal{I}$ of ρ, we have that*

$$\rho \models \varphi \textit{ iff } I \models fol_p(\varphi, last)$$

where $last = length(\rho) - 1$.

2.5.3 MONA and FOL-to-MONA Encoding

In the following, firstly we introduce the MONA tool highlighting its main features, how it works and what is its role in this thesis. Secondly, we concentrate on the MONA syntax and we describe the algorithm to translate a FOL formula into a MONA program.

MONA

MONA (Elgaard et al., 1998) is a sophisticated tool written in C/C++ for the construction of symbolic DFA from logical specifications. This tool has been implemented starting from 1997 from the BRICS (a research center in computer science located at the Aarhus University) with the aim of efficiently implementing decision procedures for the *Weak Second-order Theory of One or Two successors* (WS1S/WS2S). These two theories are also called monadic (from here the name of the tool) second-order logics and are decidable[3] since allowed second-order variables are interpreted as a finite set of numbers. Moreover, the WS1S theory is a fragment of arithmetic augmented with second-order quantification over finite sets of natural numbers. Indeed, first-order terms represents just natural numbers. Furthermore, WS1S has not the addition operator because that would make the theory undecidable, however there is the unary predicate $+1$ that stands for the successor function. On the other hand, WS2S is a generalization of WS1S to tree structures. Hence, MONA efficiently translates WS1S and WS2S formulas respectively into minimum DFAs and GTAs (Guided Tree Automata (Biehl et al., 1996)), representing them by shared, multi-terminal BDDs (Binary Decision Diagrams (Henriksen et al., 1995)). Having considered the polyedric features of MONA, we will only use the translation to DFAs.

MONA has a lot of possible applications that have been published during the years. Additionally, thanks to its APIs, it could be used both as a standalone tool and as an integrated tool for other programs. Some examples of MONA usage are the following:

- Hardware verification
- Controller systems
- Program and Protocol verification
- Software Engineering

At this point, we can explain how MONA works, at least for the part related to the DFA construction from a FOL formula. However, before doing that, we would like to clarify what the exact role of MONA is within this thesis. As stated before and as we will see in Chapter 3, MONA has been employed as a tool that translates a monadic FOL formula on finite linearly ordered domains, encoded as a M2L-Str[4], into a minimum DFA.

Now, we can briefly describe how MONA works.

[3] A logic is decidable if there exists an algorithm such that for any given formula it determines its truth value.

[4] M2L-Str is a slight variation of WS1S where formulas are interpreted over *finite string* models, rather than *infinite string* models

FOL-to-MONA Encoding

The MONA syntax is quite similar to the WS1S syntax, but it has its own method to define variables and it has been enhanced with some special details, also known as syntactic sugar, making the overall language more readable and allowing to express things more clearly and more concisely.

MONA is executed on a file, with *.mona* extension, in which we can find some declarations and WS1S/WS2S formulas. We will refer to such file as the *.mona* program, henceforth. After the execution of the tool with a *.mona* program, we get a DFA. Additionally, MONA carries out an analysis of the program by recognizing the set of satisfying interpretations for the program. Let us consider the following example (Klarlund and Møller, 2001):

Example 2.6. A `simple.mona` program:

```
var2 P,Q;
P\Q = {0,4} union {1,2};
```

First, we have declared `P` and `Q` as second-order variables. After that, we have defined a formula telling that the set difference between `P` and `Q` is the union of set `{0,4}` and `{1,2}`. Obviously, this formula is not always true, nonetheless there is an interpretation that satisfies it. For instance, the assignments `{0,1,2,4}` to `P` and `{5}` to `Q`. This interpretation can also be represented as a bit string for each variable, where positions in the string correspond to natural numbers, 1 means that the number is in the set (remember that a second-order variable is a set) whereas 0 means that is not. In this case, we would have `P`$\rightarrow$ 111010 and `Q`$\rightarrow$ 000001. Thus, it is possible to define a *language* associated to these bit strings and, since it is *regular*, it is also possible to build a DFA. Moreover, MONA assumes that all defined formulas in the program are in conjunct and each statement should be terminated by a semicolon. There are also additional elements consisting the MONA syntax depicted in Figure 2.1. As we can see from that Figure, there are also quantifiers and all usual logical connectives (i.e. those used in FOL). In addition, since we would like to write FOL on *finite linearly ordered domains*, we should enable the M2L-Str mode specifying `m2l-str;` at the beginning of the MONA program. Actually, `m2l-str;` is a shortcut for:

```
1 ws1s;
2 var2 $ where ~ex1 p where true: p notin $ & p+1 in $;
3 allpos $;
4 defaultwhere1(p) = p in $;
5 defaultwhere2(P) = P sub $;
```

At the first line, it is declared the intent to use exclusively WS1S. Then, at line 2, there is the declaration of a second-order variable `$` ensuring it to always have the value $\{0, \ldots, n-1\}$ for some n. Likewise, it is needed the declaration at line 3 to bound the domain of interest. Lastly, at lines 4 and 5, the program restrict all first- and second-order variables to `$`.

Numbers (1st order terms)

0	`0`
$+$	`+`
$-$	`-`

Sets (2nd order terms)

$\emptyset$	`empty`
$\cup$	`union`
$\cap$	`inter`
$\setminus$	`\`
$\{\ldots\}$	`{...}`

Formulas (0th order terms)

0th order arguments	
$\neg$	`~`
$\wedge$	`&`
$\vee$	`|`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\exists$	`ex0 ex1 ex2`
$\forall$	`all0 all1 all2`

Formulas (0th order terms)

1st order arguments	
$<$	`<`
$>$	`>`
$\leq$	`<=`
$\geq$	`>=`
$=$	`=`
$\neq$	`~=`
2nd order arguments	
$\subseteq$	`sub`
$=$	`=`
$\neq$	`~=`
1st/2nd order arguments	
$\in$	`in`
$\notin$	`notin`

Figure 2.1. The essential MONA syntax.

At this point, since we have illustrated all the necessary stuff for the translation, we are able to give the FOL-to-MONA encoding with some examples.

To begin with, all usual logic operators can be encoded following the table in Figure 2.1. Secondly, to encode the *succ* and *prev* monadic predicates, respectively defined in Equations 2.15 and 2.16, we use the successor and predecessor built-in operators as follows:

$$succ(x, y) \doteq \texttt{y=x+1}$$
$$prev(x, y) \doteq \texttt{y=x-1}$$

Additionally, the two constants 0 and *last*, already defined in 2.15, are encoded as `0` and `max($)`, respectively. Thirdly, to express existential and universal quantifiers we use the corresponding syntax as follows:

$$\exists p. \doteq \texttt{ex1 p:}$$
$$\forall p. \doteq \texttt{all1 p:}$$

Then, we can express first-order predicates symbols with set containment. For instance, if we have $A(x)$, before we must declare it as `var2 A;` and, then, encode it as `x in A`, whereas its negation ($\neg A(x)$) would be `x notin A`. Finally, *true* and *false* remain the same. In the following, we give some examples.

Example 2.7. FOL-to-MONA encoding examples:

- Suppose we have the LTL$_f$ formula $\Diamond G$, its translation to FOL according to Definition 2.10 is:
$$\exists y.0 \leq y \leq \mathit{last} \wedge G(y) \tag{2.17}$$
(we have not included the last part $\forall z.0 \leq z < y \Rightarrow \mathit{true}$ since it is trivially *true*). The MONA program corresponding to the formula in 2.17 is the following:

```
m2l-str;
var2 G;
ex1 y: 0<=y & y<=max($) & y in G;
```

- Suppose we have the LTL$_f$ formula $\Box G$, its translation to FOL according to Definition 2.10 is:
$$\neg(\exists y.0 \leq y \leq \mathit{last} \wedge \neg G(y)) \tag{2.18}$$
The MONA program corresponding to the formula in 2.18 is the following:

```
m2l-str;
var2 G;
~(ex1 y: 0<=y & y<=max($) & y notin G);
```

- Suppose we have the PLTL formula $A \mathcal{S} B)$, its translation to FOL according to Definition 2.11 is:
$$\exists y.0 \leq y \leq \mathit{last} \wedge B(y) \wedge \forall z.y < z \leq \mathit{last} \Rightarrow A(z) \tag{2.19}$$
The MONA program corresponding to the formula in 2.19 is the following:

```
m2l-str;
var2 A,B;
(ex1 y: 0<=y & y<=max($) & y in B & (all1 z: y<z & z<=max($) => z in A));
```

2.6 Summary

In this chapter, we have illustrated the theoretical framework, consisting of LTL, LTL$_f$ and PLTL formalisms, underlying the thesis. These formal languages have been described focusing the attention on their syntax, semantics and interesting properties. Besides, we have talked about the theory behind the translation procedure of LTL$_f$ and PLTL formulas to DFAs. Finally, we have presented the MONA tool explaining in details the encoding process starting from an LTL$_f$/PLTL formula to a MONA program passing through a FOL translation.

Chapter 3

LTL$_f$2DFA

In this chapter we will present LTL$_f$2DFA, a software package written in Python.

3.1 Introduction

LTL$_f$2DFA is a Python tool that processes a given LTL$_f$/PLTL formula and generates the corresponding minimized DFA using MONA (Elgaard et al., 1998). In addition, it offers the possibility to compute the DFA with or without the DECLARE assumption (De Giacomo et al., 2014). The main features provided by the library are:

- parsing an LTL$_f$/PLTL formula;
- translation of an LTL$_f$/PLTL formula to MONA program;
- conversion of an LTL$_f$/PLTL formula to DFA automaton.

LTL$_f$2DFA can be used with Python>=3.6 and has the following dependencies:

- PLY, a pure-Python implementation of the popular compiler construction tools Lex and Yacc. It has been employed for parsing the input LTL$_f$ formula;
- MONA, a C++ tool that translates formulas to DFA. It has been used for the generation of the DFA;
- Dotpy, a Python library able to parse and modify `.dot` files. It has been utilized for post-processing the MONA output.

The package is available to download on PyPI and you can install it by typing in the terminal:

```
pip install ltlf2dfa
```

All the code is available online on GitHub[1], it is open source and it is released under the MIT License. Moreover, LTL$_f$2DFA can also be tried online at ltlf2dfa.diag.uniroma1.it.

[1] https://github.com/Francesco17/LTLf2DFA

3.2 Package Structure

The structure of the LTL$_f$2DFA package is quite simple. It consists of a main folder called `ltlf2dfa/` which hosts the most important library's modules:

- `Lexer.py`, where the Lexer class is defined;
- `Parser.py`, where the Parser class is defined;
- `Translator.py`, where the main APIs for the translation are defined;
- `DotHandler.py`, where we the MONA output is post-processed.

In the following paragraphs we will explore each module in detail.

3.2.1 Lexer.py

In the `Lexer.py` module we can find the declaration of the `MyLexer` class which is in charge of handling the input string and tokenizing it. Indeed, it implements a tokenizer that splits the input string into declared individual tokens. To our extent, we have defined the class as in Listing 3.1

Listing 3.1. `Lexer.py` module

```python
import ply.lex as lex

class MyLexer(object):

    reserved = {
        'true':    'TRUE',
        'false':   'FALSE',
        'X':       'NEXT',
        'W':       'WEAKNEXT',
        'R':       'RELEASE',
        'U':       'UNTIL',
        'F':       'EVENTUALLY',
        'G':       'GLOBALLY',
        'Y':       'PASTNEXT', #PREVIOUS
        'S':       'PASTUNTIL', #SINCE
        'O':       'PASTEVENTUALLY', #ONCE
        'H':       'PASTGLOBALLY' #HISTORICALLY
    }
    # List of token names. This is always required
    tokens = (
        'TERM',
        'NOT',
        'AND',
```

```
        'OR',
        'IMPLIES',
        'DIMPLIES',
        'LPAR',
        'RPAR'
    ) + tuple(reserved.values())

    # Regular expression rules for simple tokens
    t_TRUE = r'true'
    t_FALSE = r'false'
    t_AND = r'\&'
    t_OR = r'\|'
    t_IMPLIES = r'\->'
    t_DIMPLIES = r'\<->'
    t_NOT = r'\~'
    t_LPAR = r'\('
    t_RPAR = r'\)'
    # FUTURE OPERATORS
    t_NEXT = r'X'
    t_WEAKNEXT = r'W'
    t_RELEASE = r'R'
    t_UNTIL = r'U'
    t_EVENTUALLY = r'F'
    t_GLOBALLY = r'G'
    # PAST OPERATOR
    t_PASTNEXT = r'Y'
    t_PASTUNTIL = r'S'
    t_PASTEVENTUALLY = r'O'
    t_PASTGLOBALLY = r'H'

    t_ignore = r'␣'+'\n'

    def t_TERM(self, t):
        r'(?<![a-z])(?!true|false)[_a-z0-9]+'
        t.type = MyLexer.reserved.get(t.value, 'TERM')
        return t # Check for reserved words

    def t_error(self, t):
        print("Illegal␣character␣'%s'␣in␣the␣input␣formula" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    def build(self,**kwargs):
```

```
67        self.lexer = lex.lex(module=self, **kwargs)
```

Firstly, we have defined the reserved words within a dictionary so to match each reserved word with its identifier. Secondly, we have defined the tokens list with all possible tokens that can be produced by the lexer. This tokens list is always required for the implementation of a lexer. Then, each token has to be specified by writing a regular expression rule. If the token is simple it can be specified using only a string. Otherwise, for non trivial tokens we have to write the regular expression in a class method as for our token `TERM` in line 56. In that case, defining the token rule as a method is also useful when we would like to perform other actions. After that, we have a method to handle unrecognized tokens and, finally, we have written the function that builds the lexer.

3.2.2 Parser.py

In the `Parser.py` module we can find the declaration of `MyParser` class which implements the parsing component of `PLY`. The `MyParser` class operates after the Lexer has split the input string into known tokens. The main feature of the parser is to interpret and build the appropriate data structure for the given input. To this extent, the most important aspect of a parser is the definition of the *syntax*, usually specified in terms of a BNF[2] grammar, that should be unambiguous. Furthermore, `Yacc`, the parsing component of `PLY`, implements a parsing technique known as LR-parsing or shift-reduce parsing. In particular, this parsing technique works on a bottom up fashion that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side and so on until there is no more rule to apply. The parser implementation is shown in Listing 3.2

Listing 3.2. `Parser.py` module

```python
import ply.yacc as yacc
from ltlf2dfa.Lexer import MyLexer

class MyParser(object):

    def __init__(self):
        self.lexer = MyLexer()
        self.lexer.build()
        self.tokens = self.lexer.tokens
        self.parser = yacc.yacc(module=self)
        self.precedence = (

            ('nonassoc', 'LPAR', 'RPAR'),
            ('left', 'AND', 'OR', 'IMPLIES', 'DIMPLIES', 'UNTIL', \
              'RELEASE', 'PASTUNTIL'),
```

[2]The Backus–Naur form is a notation technique for context-free grammars.

```
            ('right', 'NEXT', 'WEAKNEXT', 'EVENTUALLY', \
            'GLOBALLY', 'PASTNEXT', 'PASTEVENTUALLY', 'PASTGLOBALLY'),
            ('right', 'NOT')
        )

    def __call__(self, s, **kwargs):
        return self.parser.parse(s, lexer=self.lexer.lexer)

    def p_formula(self, p):
        '''
        formula : formula AND formula
                | formula OR formula
                | formula IMPLIES formula
                | formula DIMPLIES formula
                | formula UNTIL formula
                | formula RELEASE formula
                | formula PASTUNTIL formula
                | NEXT formula
                | WEAKNEXT formula
                | EVENTUALLY formula
                | GLOBALLY formula
                | PASTNEXT formula
                | PASTEVENTUALLY formula
                | PASTGLOBALLY formula
                | NOT formula
                | TRUE
                | FALSE
                | TERM
        '''

        if len(p) == 2: p[0] = p[1]
        elif len(p) == 3:
            if p[1] == 'F': # F(a) == true UNITL A
                p[0] = ('U','true', p[2])
            elif p[1] == 'G': # G(a) == not(eventually (not A))
                p[0] = ('~',('U', 'true', ('~',p[2])))
            elif p[1] == 'O': # O(a) = true SINCE A
                p[0] = ('S','true', p[2])
            elif p[1] == 'H': # H(a) == not(pasteventually(not A))
                p[0] = ('~',('S', 'true', ('~',p[2])))
            else:
                p[0] = (p[1], p[2])
        elif len(p) == 4:
```

```
            if p[2] == '->':
                p[0] = ('|', ('~', p[1]), p[3])
            elif p[2] == '<->':
                p[0] = ('&', ('|', ('~', p[1]), p[3]), ('|', ('~', p[3]),\
                p[1]))
            else:
                p[0] = (p[2],p[1],p[3])
        else: raise ValueError


    def p_expr_group(self, p):
        '''
        formula : LPAR formula RPAR
        '''
        p[0] = p[2]

    def p_error(self, p):
        raise ValueError("Syntax␣error␣in␣input!␣%s" %str(p))
```

As we can see, as soon as the parser is instantiated it builds the lexer, gets the tokens and defines their precedence if needed. Then, we have defined methods of the `MyParser` class that are in charge of constructing the syntax tree structure from tokens found by the lexer in the input string. In our case, we have chosen to use as data structure a tuple of tuples as it is the one of the simplest data structure in Python. In general, a tuple of tuples represents a tree where each node represents an item present in the formula.

For instance, the LTL$_f$ formula $\varphi = G(a \Rightarrow Xb)$ is represented as $('\sim', ('U', '\mathit{true}', ('\sim', ('|', ('\sim', 'a'), ('X', 'b')))))$ and it corresponds to a tree as the one depicted in Figure 3.1. Finally, as in the `MyLexer` class, we have to handle errors defining a specific method.

LTL$_f$2DFA can be used just for the parsing phase of an LTL$_f$/PLTL formula as shown in Listing 3.3.

Listing 3.3. How to use only the parsing phase of LTL$_f$2DFA.

```
from ltlf2dfa.Parser import MyParser

formula = "G(a->Xb)"
parser = MyParser()
parsed_formula = parser(formula)

print(parsed_formula) # syntax tree as tuple of tuples
```

3.2.3 Translator.py

The `Translator.py` module contains the majority of APIs that the LTL$_f$2DFA package exposes. Indeed, this module consists of a `Translator` class which concerns the core

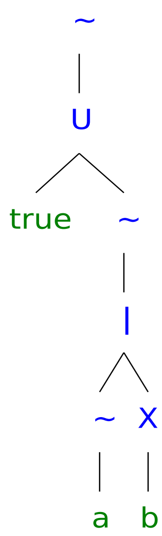


Figure 3.1. The syntax tree generated for the formula "$G(a \Rightarrow Xb)$". Symbols are in green while operators are in blue.

feature of the package: the translation of an LTL$_f$/PLTL formula into the corresponding minimum DFA. Since the package takes advantage of the MONA tool for the formula conversion, the `Translator` class has to translate first the given formula into the syntax recognized by MONA, then create the input program for MONA and, finally, invoke MONA to get back the resulting DFA in the Graphviz[3] format. The main methods of the `Translator` class are:

- `translate()`, which starting from the formula syntax tree generated (Figure 3.1) in the parsing phase translates it into a string using the syntax of MONA;
- `createMonafile(flag)`, which, as the name suggests, creates the program *.mona* that will be given as input to MONA. The flag parameter is going to be `True` of `False` whether we need to compute also DECLARE assumptions or not;
- `invoke_mona()`, which invokes MONA in order to obtain the DFA.

Now we will go into details of the methods stated above showing their implementation.

The `translate` method

The `translate` method is a crucial step towards reaching a good result and performance. Formally, the translation procedure from an LTL$_f$/PLTL formula to the MONA syntax is

[3]Graphviz is open source graph visualization software. For further details see https://www.graphviz.org

done passing through FOL as shown in 3.1.

$$\text{LTL}_f/\text{PLTL} \rightarrow \text{FOL} \rightarrow \text{MONA} \tag{3.1}$$

The former translation from LTL$_f$/PLTL to FOL is done accordingly to (De Giacomo and Vardi, 2013), while the latter follows from (Klarlund and Møller, 2001), as seen in Section 2.5. In Listing 3.4 we can see the translation's implementation. Three dots ... represent omitted code.

Listing 3.4. The `translate` method.

```
import ...

class Translator:
    ...

    def translate(self):
        self.translated_formula = translate_bis(self.parsed_formula, \
        self.formulaType, var='v_0')+";\n"

   ...

def translate_bis(formula_tree, _type, var):
    if type(formula_tree) == tuple:
        if formula_tree[0] == '&':
            if var == 'v_0':
                if _type == 2:
                    a = translate_bis(formula_tree[1], _type, 'max($)')
                    b = translate_bis(formula_tree[2], _type, 'max($)')
                else:
                    a = translate_bis(formula_tree[1], _type, '0')
                    b = translate_bis(formula_tree[2], _type, '0')
            else:
                a = translate_bis(formula_tree[1], _type, var)
                b = translate_bis(formula_tree[2], _type, var)
            if a == 'false' or b == 'false':
                return 'false'
            elif a == 'true':
                if b == 'true': return 'true'
                else: return b
            elif b == 'true': return a
            else: return '('+a+' & '+b+')'
        elif formula_tree[0] == '|':
            if var == 'v_0':
                if _type == 2:
```

```
                    a = translate_bis(formula_tree[1], _type, 'max($)')
                    b = translate_bis(formula_tree[2], _type, 'max($)')
                else:
                    a = translate_bis(formula_tree[1], _type, '0')
                    b = translate_bis(formula_tree[2], _type, '0')
            else:
                a = translate_bis(formula_tree[1], _type, var)
                b = translate_bis(formula_tree[2], _type, var)
            if a == 'true' or b == 'true':
                return 'true'
            elif a == 'false':
                if b == 'true': return 'true'
                elif b == 'false': return 'false'
                else: return b
            elif b == 'false': return a
            else: return '('+a+' | '+b+')'
        elif formula_tree[0] == '~':
            if var == 'v_0':
                if _type == 2:
                    a = translate_bis(formula_tree[1], _type, 'max($)')
                else:
                    a = translate_bis(formula_tree[1], _type, '0')
            else: a = translate_bis(formula_tree[1], _type, var)
            if a == 'true': return 'false'
            elif a == 'false': return 'true'
            else: return '~('+ a +')'
        elif formula_tree[0] == 'X':
            new_var = _next(var)
            a = translate_bis(formula_tree[1], _type, new_var)
            if var == 'v_0':
                return '('+ 'ex1 '+new_var+': '+ new_var +' = 1 '+ '& '+ \
                a +')'
            else:
                return '('+ 'ex1 '+new_var+': '+ new_var +' = '+ var + \
                ' + 1 '+ '& '+ a +')'
        elif formula_tree[0] == 'U':
            new_var = _next(var)
            new_new_var = _next(new_var)
            a = translate_bis(formula_tree[2], _type, new_var)
            b = translate_bis(formula_tree[1], _type, new_new_var)

            if var == 'v_0':
                if b == 'true': return '( '+ 'ex1 '+new_var+': 0 <= '+ \
```

```
                new_var+'␣&␣'+ new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+ \
                new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+ \
                new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
                elif a == 'false': return 'false'
                else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+new_var+ \
                '␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣&␣all1␣'+ \
                new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+ \
                new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
            else:
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+ \
                '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+ \
                '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+ \
                new_new_var+':␣'+var+'␣<=␣'+new_new_var+'␣&␣'+ \
                new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
                elif a == 'false': return 'false'
                else: return '(␣'+ 'ex1␣'+new_var+':␣'+var+'␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a + \
                '␣&␣all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<␣'+new_var+'␣=>␣'+b+'␣)'
        elif formula_tree[0] == 'W':
            new_var = _next(var)
            a = translate_bis(formula_tree[1], _type, new_var)
            if var == 'v_0':
                return '(0␣=␣max($))␣|␣('+ 'ex1␣'+new_var+':␣'+new_var+\
                '␣=␣1␣'+ '&␣'+ a +')'
            else:
                return '('+ var +'␣=␣max($))␣|␣('+ 'ex1␣'+new_var+':␣'+\
                new_var +'␣=␣'+ var + '␣+␣1␣'+ '&␣'+ a +')'

        elif formula_tree[0] == 'R':
            new_var = _next(var)
            new_new_var = _next(new_var)
            a = translate_bis(formula_tree[2], _type, new_new_var)
            b = translate_bis(formula_tree[1], _type, new_var)

            if var == 'v_0':
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'\
                +new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+\
                new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
```

```
                '(all1␣'+new_new_var+':␣0␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+\
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+b+')'
                elif b == 'false': return '(all1␣'+new_new_var+':␣0␣<=␣'+\
                new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
                else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+new_var+\
                '␣&␣'+new_var+'␣<=␣max($)␣&␣'+ b +'␣&␣all1␣'+\
                new_new_var+':␣0␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
                '(all1␣'+new_new_var+':␣0␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
            else:
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+\
                '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣all1␣'+\
                new_new_var+':␣'+var+'␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
                '(all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+'␣&␣'+\
                new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+':␣'+var+\
                '␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+b+')'
                elif b == 'false': return '(all1␣'+new_new_var+':␣'+\
                var+'␣<=␣'+new_new_var+'␣&␣'+new_new_var+\
                '␣<=␣max($)␣=>␣'+a+'␣)'
                else: return '(␣'+ 'ex1␣'+new_var+':␣'+var+'␣<=␣'+\
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ b +\
                '␣&␣all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣'+new_var+'␣=>␣'+a+'␣)␣|'\
                '(all1␣'+new_new_var+':␣'+var+'␣<=␣'+new_new_var+\
                '␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+a+'␣)'
        elif formula_tree[0] == 'Y':
            new_var = _next(var)
            a = translate_bis(formula_tree[1], _type, new_var)
            if var == 'v_0':
                return '('+ 'ex1␣'+new_var+':␣'+ new_var + \
                '␣=␣max($)␣-␣1␣'+ '&␣max($)␣>␣0␣&␣'+ a +')'
            else:
                return '('+ 'ex1␣'+new_var+':␣'+ new_var + \
                '␣=␣'+ var + '␣-␣1␣'+ '&␣'+new_var+'␣>␣0␣&␣'+ a +')'
        elif formula_tree[0] == 'S':
            new_var = _next(var)
            new_new_var = _next(new_var)
            a = translate_bis(formula_tree[2], _type, new_var)
```

```
            b = translate_bis(formula_tree[1], _type, new_new_var)

            if var == 'v_0':
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+ \
                ':␣0␣<=␣'+new_var+'␣&␣'+new_var+ \
                '␣<=␣max($)␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
                new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+b+'␣)'
                elif a == 'false': return 'false'
                else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a + \
                '␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
                new_new_var+'␣&␣'+new_new_var+'␣<=␣max($)␣=>␣'+b+'␣)'
            else:
                if b == 'true': return '(␣'+ 'ex1␣'+new_var+ \
                ':␣0␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣max($)␣&␣'+ a +'␣)'
                elif a == 'true': return '(␣'+ 'ex1␣'+new_var+ \
                ':␣0␣<=␣'+new_var+'␣&␣'+new_var+'␣<=␣'+var+ \
                '␣&␣all1␣'+new_new_var+':␣'+new_var+'␣<␣'+ \
                new_new_var+'␣&␣'+new_new_var+'␣<=␣'+var+'␣=>␣'+b+'␣)'
                elif a == 'false': return 'false'
                else: return '(␣'+ 'ex1␣'+new_var+':␣0␣<=␣'+ \
                new_var+'␣&␣'+new_var+'␣<=␣'+var+'␣&␣'+ a +'␣&␣all1␣'+ \
                new_new_var+':␣'+new_var+'␣<␣'+new_new_var+'␣&␣'+ \
                new_new_var+'␣<=␣'+var+'␣=>␣'+b+'␣)'
    else:
        # handling non-tuple cases
        if formula_tree == 'true': return 'true'
        elif formula_tree == 'true': return 'false'

        # BASE CASE OF RECURSION
        else:
            if var == 'v_0':
                if _type == 2:
                    return 'max($)␣in␣'+ formula_tree.upper()
                else:
                    return '0␣in␣'+ formula_tree.upper()
            else:
                return var + '␣in␣' + formula_tree.upper()

def _next(var):
    if var == '0' or var == 'max($)': return 'v_1'
```

```
207     else:
208         s = var.split('_')
209         s[1] = str(int(s[1])+1)
210         return '_'.join(s)
```

As we can see, the `translate` method is actually very simple. In fact, it just calls the `translate_bis` function (line 12) to perform the proper translation. The function works in a recursive fashion taking as input the parsed formula and a variable and outputting a string containing the result. Obviously, when an instance of the `Translator` class is created the input formula is checked to have either only future or past operators. The base case of the recursion handles the translation of symbols as they are the leaves of the syntax tree composed in the parsing phase (Figure 3.1). On the other hand, the recursive step regards the handling of operators (non leaf components of the syntax tree) which are in our case $\wedge$, $\vee$, $\neg$, $\circ$, $\mathcal{U}$, $\ominus$, $\mathcal{S}$. During the translation, we simplify the resulting formula by avoiding pieces of the expression that are logically `True` or `False`. This simplification has two main advantages. First, it substantially reduces the length of the resulting formula, improving its readability. Second, it increases the computation performances of MONA. Additionally, since the MONA syntax requires the declaration of the free variables, the `translate_bis` function has to compute also the appriopriate free variables declaration. In this terms, the translation function uses the `_next` function to compute the next variable each time is needed.

The `createMonafile` method

The `createMonafile` method is employed to write the program *.mona* and save it in the main directory. It takes as input a boolean flag that, as stated before, stands for indicating whether one would like to compute and add the DECLARE assumption or not. In particular, in formal logic, as stated in (De Giacomo et al., 2014), the DECLARE assumption is expressed as in 3.2.

$$\Box(\bigvee_{a \in \mathcal{P}} a) \wedge \Box(\bigwedge_{a,b \in \mathcal{P}, a \neq b} a \Rightarrow \neg b) \tag{3.2}$$

It consists essentially in two parts joined by the $\wedge$ operator. The former indicates that it is always true that at each point in time only one symbol is *true*, while the latter means that always for each couple of different symbols in the formula if one is *true* the other must be *false*. The practical part can be seen in Listing 3.5.

Listing 3.5. The `createMonafile` method.

```
...
    def compute_declare_assumption(self):
        pairs = list(it.combinations(self.alphabet, 2))

        if pairs:
            first_assumption = "~(ex1 y: 0<=y & y<=max($) & ~("
```

```
            for symbol in self.alphabet:
                if symbol == self.alphabet[-1]: first_assumption += \
                'y␣in␣'+ symbol +'))'
                else : first_assumption += 'y␣in␣'+ symbol +'␣|␣'

            second_assumption = "~(ex1␣y:␣0<=y␣&␣y<=max($)␣&␣~("
            for pair in pairs:
                if pair == pairs[-1]: second_assumption += '(y␣notin␣'+ \
                pair[0]+'␣|␣y␣notin␣'+pair[1]+ ')));'
                else: second_assumption += '(y␣notin␣'+ pair[0]+ \
                '␣|␣y␣notin␣'+pair[1]+ ')␣&␣'

            return first_assumption +'␣&␣'+ second_assumption
        else:
            return None

    def buildMonaProgram(self, flag_for_declare):
        if not self.alphabet and not self.translated_formula:
            raise ValueError
        else:
            if flag_for_declare:
                if self.compute_declare_assumption() is None:
                    if self.alphabet:
                        return self.headerMona + \
                        'var2␣' + ",␣".join(self.alphabet) + ';\n' + \
                         self.translated_formula
                    else:
                        return self.headerMona + self.translated_formula
                else: return self.headerMona + 'var2␣' +\
                 ",␣".join(self.alphabet) + ';\n' + \
                 self.translated_formula + \
                 self.compute_declare_assumption()
            else:
                if self.alphabet:
                    return self.headerMona + 'var2␣' +\
                     ",␣".join(self.alphabet) + ';\n' + \
                     self.translated_formula
                else:
                    return self.headerMona + self.translated_formula

    def createMonafile(self, flag):
        program = self.buildMonaProgram(flag)
        try:
```

```
50             with open('./automa.mona', 'w+') as file:
51                 file.write(program)
52                 file.close()
53         except IOError:
54             print('Problem␣with␣the␣opening␣of␣the␣file!')
55 ...
```

As shown in the code, the `createMonafile` method calls another method, the `buildMonaProgram` (line 23), which literally builds the *.mona* program by joining all pieces that should belong to it. Instead, regarding the DECLARE assumption, if needed, it is added to the *.mona* program directly translated through `compute_declare_assumption` method at line 2.

The `invoke_mona` method

Finally, the `invoke_mona` method is the one that executes the MONA compiled executable giving it the *.mona* program. Consequently, the DFA resulting from the computation of MONA will be stored in the main directory. As stated in 3.1, the LTL$_f$2DFA package requires MONA to be installed. Indeed, without this requirements the `invoke_mona` method will raise an error. The implementation can be seen in Listing 3.6.

Listing 3.6. The `invoke_mona` method.

```
...
    def invoke_mona(self, path='./inter-automa'):
        if sys.platform == 'linux':
            package_dir = os.path.dirname(os.path.abspath(__file__))
            mona_path = pkg_resources.resource_filename('ltlf2dfa','mona')
            if os.access(mona_path, os.X_OK): #check if mona is executable
                try:
                    subprocess.call(package_dir+'/./mona␣-u␣-gw␣' + \
                    './automa.mona␣>␣' + path + '.dot', shell=True)
                except subprocess.CalledProcessError as e:
                    print(e)
                    exit()
                except OSError as e:
                    print(e)
                    exit()
            else:
                print('[ERROR]:␣MONA␣tool␣is␣not␣executable...')
                exit()
        else:
            try:
                subprocess.call('mona␣-u␣-gw␣./automa.mona␣>␣' + path + \
                '.dot', shell=True)
            except subprocess.CalledProcessError as e:
```

```
                print(e)
                exit()
            except OSError as e:
                print(e)
                exit()
...
```

To the execute of the MONA tool we have leveraged the built-in module `subprocess` that enables to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

Unfortunately, the DFA resulting from MONA needs to be post-processed because of some extra states added for other purposes not relevant for us. This aspect will be better explained in the following subsection 3.2.4.

3.2.4 DotHandler.py

The `DotHandler` class has been created in order to manage separately and better the post-processing of the DFA, in *.dot* format, resulting from the computation of MONA. Indeed, since MONA has been developed for different purposes, its output has an additional initial state and transition that to our intent are completely meaningless.

Additionally, the interaction with the *.dot* format has been implemented thanks to the `dotpy` library (available on GitHub[4]) developed for this specific purpose paying particular attention to performances.

As we can see in the implementation of the `DotHandler` class in Listing 3.7, the main methods are `modify_dot` and `output_dot`.

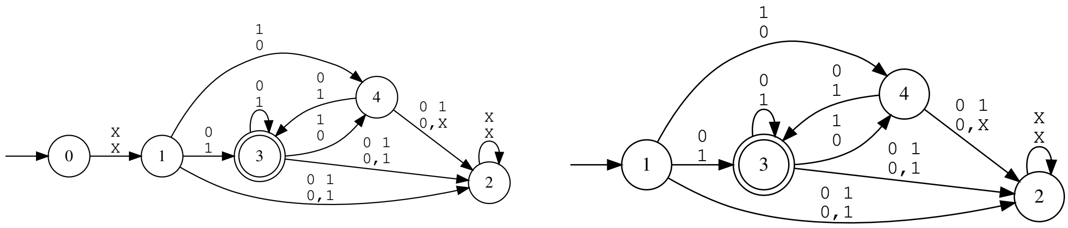
Listing 3.7. The `DotHandler` class.

```
from dotpy.parser.parser import MyParser
import os

class DotHandler:

    def __init__(self, path='./inter-automa.dot'):
        self.dot_path = path
        self.new_digraph = None

    def modify_dot(self):
        if os.path.isfile(self.dot_path):
            parser = MyParser()
            with open(self.dot_path, 'r') as f:
                dot = f.read()
                f.close()

```

[4] https://github.com/Francesco17/dotpy

```
17            graph = parser(dot)
18            if not graph.is_singleton():
19                graph.delete_node('0')
20                graph.delete_edge('init', '0')
21                graph.delete_edge('0', '1')
22                graph.add_edge('init', '1')
23            else:
24                graph.delete_edge('init', '0')
25                graph.add_edge('init', '0')
26            self.new_digraph = graph
27        else:
28            print('[ERROR] - No file DOT exists')
29            exit()
30
31    def delete_intermediate_automaton(self):
32        if os.path.isfile(self.dot_path):
33            os.remove(self.dot_path)
34            return True
35        else:
36            return False
37
38    def output_dot(self, result_path='./automa.dot'):
39        try:
40            if self.delete_intermediate_automaton():
41                with open(result_path, 'w+') as f:
42                    f.write(str(self.new_digraph))
43                    f.close()
44            else:
45                raise IOError('[ERROR] - Something wrong occurred in '+ \
46                'the elimination of intermediate automaton.')
47        except IOError:
48            print('[ERROR] - Problem with the opening of the file %s!' \
49            %result_path)
```

The former method at line 10 takes advantage of the APIs exposed by `dotpy`. Especially, it parses the *.dot* file output of MONA (Figure 3.2a), deletes the starting node 0 and the edge from node 0 to node 1 and, finally, makes node 1 initial. Consequently, the latter method at line 38 manages the output of the final post-processed DFA (Figure 3.2b) and stores it in the main directory. For instance, in Figure 3.2 we can see graphically what is the outcome of the post-processing of the automaton corresponding to the formula $\varphi = \Box(a \Rightarrow \bigcirc b)$.



(a) The DFA output from MONA

(b) The DFA post-processed

Figure 3.2. Before and after DFA post-processing

3.3 Interpreting LTL$_f$2DFA output

In this section, we explain through examples how to interpret and read the output DFA resulting from the LTL$_f$2DFA computation.

To begin with, circle nodes represents automaton states and doubled circle nodes represents those state that are accepting or final for the automaton. Labels on transitions stand for all possible values of formula symbols. A specific formula symbol in a transition must have one of the following values:

- **1**: means that the formula symbol is *true* in that transition;
- **0**: means that the formula symbol is *false* in that transition;
- **X**: means *don't care*, i.e. the formula symbol can be both *true* or *false* in that transition. In other words, it means that the transition can be done no matter is the actual value of the formula symbol.

Finally, when a formula has multiple symbols, the value of each symbol has to be read vertically in order of symbols declaration in the formula. In the following, we will give some examples.

Example 3.1. Let us consider the formula $\varphi = \Diamond g$ and its corresponding automaton depicted in Figure 3.3. The first transition without label indicates the initial state.

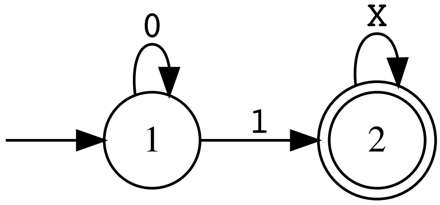


Figure 3.3. Minimum DFA for the formula $\varphi = \Diamond g$.

Then, the first loop on state 1 is done when g is *false*. Afterwards, the transition from state 1 to state 2 can be done only if g is *true*. Finally, the loop on state 2 has the label

"X" meaning that once the automaton has arrived on state 2, whatever action it does (also g and $\neg g$) it remains on state 2, which is, by the way, final for the automaton.

Example 3.2. Let us consider the formula $\varphi = \Box(a \Rightarrow \Diamond b)$ and its corresponding automaton depicted in Figure 3.4. As usual, state 1 is the starting state. However, this

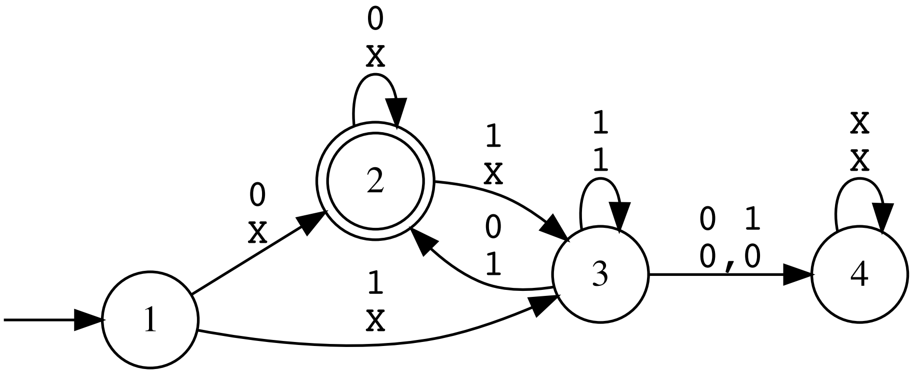


Figure 3.4. Minimum DFA for the formula $\varphi = \Box(a \Rightarrow \Diamond b)$.

case is a little bit different from the previous one. Indeed, now the formula has two symbols, namely a and b. Since the order of declaration is a, b, labels on transition has to be read vertically following this order. For instance, the label on transition from state 1 to state 2 reports $\substack{0 \\ \mathtt{X}}$ meaning that the automaton can walk this transition only if a is *false* (in this case, b is *don't care*, i.e. it can assume whatever value). Additionally, another interesting transition to comment is the one that goes from state 3 to state 4. Its label reports $\substack{0\ 1 \\ 0,0}$ meaning that the automaton will do that transition only if either a and b are *false* or a is *true* and b is *false*.

3.4 Comparison with FLLOAT

In this section, we will see how LTL$_f$2DFA performs compared to FLLOAT[5], which is another Python package having the conversion of an LTL$_f$ formula to a DFA as one of its features. In particular, FLLOAT handles LTL$_f$ and LDL$_f$ (*Linear Dynamic Logic on Finite Traces*) formulas, except to PLTL ones, but it provides support for syntax, semantics and parsing of PL (*Propositional Logic*), LTL$_f$ and LDL$_f$ formal languages. Additionally, its conversion is based on a different theoretical result with respect to LTL$_f$2DFA. Especially, FLLOAT directly implements the algorithm seen in Section 2.4.1 and, therefore, it could also accept traces of length 0 (empty traces). This is not the same for LTL$_f$2DFA, since it assumes traces to have at least one element. Nevertheless, we can compare them on the generation of a DFA from an LTL$_f$ formula just by forcing FLLOAT to produce DFAs concerning traces with at least one element. This could be achieved adding *true* to FLLOAT formulas with the $\wedge$ connective. For instance, given an LTL$_f$ formula φ, FLLOAT will compute the DFA for $\psi = \mathit{true} \wedge \varphi$.

[5]https://github.com/MarcoFavorito/flloat

The time execution benchmarks between these two packages was done over a set of 13 different interesting LTL$_f$ formulas of different length. The comparison consisted of executing each package over the same set of formulas n number of times and, then, repeating the multiple execution m number of times. Thus, for each formula to be converted we obtained $n \times m$ results and, finally, we kept the minimum one (i.e. the best time execution result). After gathering the results, we can show them on a histogram where on the $x-$axis there are the LTL$_f$ formulas and on the $y-$axis there is the minimum time (in seconds) needed for the package to convert it into a DFA (Figure 3.5). In the

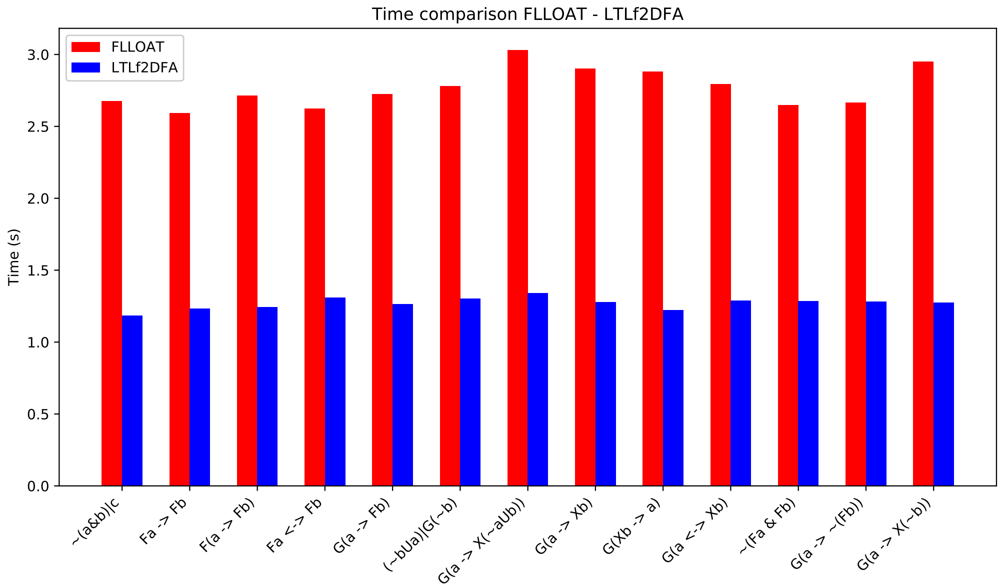


Figure 3.5. Time benchmarking of LTL$_f$2DFA wrt FLLOAT.

histogram, FLLOAT results are coloured in red, while LTL$_f$2DFA ones are depicted in blue.

As we can see from the bar chart, in FLLOAT the time needed to convert the formula increases as the length of the formula grows, whereas in LTL$_f$2DFA it is almost steady. Furthermore, it is notable that LTL$_f$2DFA is overall more than twice as fast as FLLOAT. This behaviour is due to the fact that these two packages operates in a different way. Indeed, while FLLOAT is a pure Python package, LTL$_f$2DFA uses, for the heavy task of the generation of the automaton, MONA that is written in C/C++. Hence, the real difference relies on the performance differences between C/C++ and Python programs. As a final remark, although LTL$_f$2DFA is much faster than FLLOAT, its time execution depends on the I/O system performance which can drastically reduce it. Thus, LTL$_f$2DFA results may arise depending on various factors such as disk speed, caching and filesystem.

3.5 Summary

In this chapter, we have presented the LTL$_f$2DFA Python package. We have also described the structure of the package, discussed in detail its implementation highlighting all the main features and, finally, seen how it performs in time relatively to the FLLOAT Python package.

Chapter 4

Planning for LTL$_f$/PLTL Goals

In this chapter, we will define a new approach to the problem of non-deterministic planning for extended temporal goals. In particular, we will give a solution to this problem reducing it to a *fully observable non-deterministic* (FOND) problem and taking advantage of our tool LTL$_f$2DFA, presented in Chapter 3. First of all, we will introduce the main idea and motivations supporting our approach. Then, we will give some preliminaries explaining the Planning Domain Definition Language (PDDL) language and the FOND planning problem formally. After that, we will illustrate our FOND4LTL$_f$/PLTL approach with the encoding of temporal goals into a PDDL domain and problem. Finally, we will present our practical implementation of the proposed solution.

4.1 Idea and Motivations

Planning for temporally extended goals with *deterministic* actions has been well studied during the years starting from (Bacchus and Kabanza, 1998) and (Doherty and Kvarnstram, 2001). Two main reasons why temporally extended goals have been considered over the classical goals, viewed as a desirable set of final states to be reached, are because they are not limited in what they can specify and they allows us to restrict the manner used by the plan to reach the goals. Indeed, temporal extended goals are fundamental for the specification of a collection of real-world planning problems. Yet, many of these real-world planning problems have a *non-deterministic* behavior owing to unpredictable environmental conditions. However, planning for temporally extended goals with *non-deterministic* actions is a more challenging problem and has been of increasingly interest only in recent years with (Camacho et al., 2017; De Giacomo and Rubin, 2018).

In this scenario, we have devised a solution to this problem that exploits the translation of a temporal formula to a DFA, using LTL$_f$2DFA. In particular, our idea is the following: given a non-deterministic planning problem and a temporal formula, we first obtain the corresponding DFA of the temporal formula through LTL$_f$2DFA, then, we encode such a DFA into the non-deterministic planning domain. As a result, we have reduced the original problem to a classic FOND planning problem. In other words, we compile extended temporal goals together with the original planning domain, specified in PDDL, which is suitable for input to standard (FOND) planners.

4.2 Preliminaries

In this section, we will give some basics on the PDDL specification language for domains and problems of planning and a general formalization of FOND planning.

4.2.1 PDDL

As stated before, PDDL is the acronym for Planning Domain Definition Language, which is the *de-facto* standard language for representing "classical" planning tasks. A general planning task has the following components:

- Objects: elements in the world that are of our interest;
- Predicates: objects properties that can be true or false;
- Initial state: state of the world where we start;
- Goal state: things we want to be true;
- Action/Operator: rule that changes the state.

Moreover, planning tasks are composed by two files: the *domain* file where are defined predicates and actions and a *problem* file where are defined objects, the initial state and the goal specification.

The *domain* file

The *domain* definition gives each domain a name and specifies predicates and actions available in the domain. It might also specify types, constants and other things. A simple domain has the following format:

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl] ...)
  [(:types T1 T2 T3 T4 ...)]
  (:predicates (PREDICATE_1_NAME [?A1 ?A2 ... ?AN])
               (PREDICATE_2_NAME [?A1 ?A2 ... ?AN])
              ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
   )
```

```
  (:action ACTION_2_NAME
    ...)
  ...)
```

where [] indicates optional elements. To begin with, any PDDL *domain* definition must declare its expressivity requirements given after the `:requirements` key. The basic PDDL expressivity is called STRIPS[1], whereas a more complex one is the Action Description Language (ADL), that extends STRIPS in several ways, such as providing support for negative preconditions, disjunctive preconditions, quantifiers, conditional effects etc.. Nevertheless, many planners do not support full ADL because creating plans efficiently is not trivial. Although the presence of this limitation, the PDDL language allows us to use only some of the ADL features. Furthermore, there are also other requirements often used that can be specified as `equality`, allowing the usage of the predicate `=` interpreted as equality, and `typing` allowing the typing of objects. As we will explain later in Section 4.4, our practical implementation supports, for now, only simple ADL, namely conditional effects in domain's operators which do not have any nested subformula.

Secondly, there is the predicates definition after the `:predicates` key. Predicates may have zero or more parameters variables and they specify only the number of arguments that a predicate should have. Moreover, a predicate may also have typed parameters written as `?X - TYPE_OF_X`.

Thirdly, there is a list of action definitions. An action is composed by the following items:

- *parameters*: they stand for free variables and are represented with a preceding question mark `?`;
- *precondition*: it tells when an action can be applied and, depending on given requirements, it could be differently defined (i.e. conjunctive formula, disjunctive formula, quantified formula, etc.);
- *effect*: it tells what changes in the state after having applied the action. As for the precondition, depending on given requirements, it could be differently defined (i.e. conjunctive formula, conditional formula, universally quantified formula, etc.)

In particular, in pure STRIPS domains, the precondition formula can be one of the following:

- an atomic formula as `(PREDICATE_NAME ARG1 ... ARG_N)`
- a conjunction of atomic formulas as `(and ATOM1 ... ATOM_N)`

where arguments must either be parameters of the action or constants.

If the *domain* uses the `:adl` or `:negated-precondition` an atomic formula could be expressed also as `(not (PREDICATE_NAME ARG1 ... ARG_N))`. In addition, if the domain uses `:equality`, an atomic formula may also be of the form `(= ARG1 ARG2)`.

[1] STRIPS stands for STanford Research Institute Problem Solver, which is a formal language of inputs to the homonym automated planner developed in 1971.

On the contrary, in ADL domains, a precondition formula could be one of the following:

- a general negation as `(not CONDITION_FORMULA)`
- a conjunction of condition formulas as `(and CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- a disjunction of condition formulas as `(or CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- an implication as `(imply CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- an implication as `(imply CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- a universally quantified formula as `(forall (?V1 ?V2 ...) CONDITION_FORMULA)`
- an existentially quantified formula as `(exists (?V1 ?V2 ...) CONDITION_FORMULA)`

The same division can be carried out with effects formulas. Specifically, in pure STRIPS domains, the precondition formula can be one of the following:

- an added atom as `(PREDICATE_NAME ARG1 ... ARG_N)`
- a deleted atom as `(not (PREDICATE_NAME ARG1 ... ARG_N))`
- a conjunction of effects as `(and ATOM1 ... ATOM_N)`

On the other hand, in an ADL domains, an effect formula can be expressed as:

- a conditional effect as `(when CONDITION_FORMULA EFFECT_FORMULA)`, where the `EFFECT_FORMULA` is occur only if the `CONDITION_FORMULA` holds true. A conditional effect can be placed within quantification formulas.
- a universally quantified formula as `(forall (?V1 ?V2 ...) EFFECT_FORMULA)`

As last remark that we will deepen later in Section 4.2.2, when the PDDL *domain* has *non-deterministic* actions, the effect formula of those actions expresses the non-determinism with the keyword `oneof` as `(oneof (EFFECT_FORMULA_1) ... (EFFECT_FORMULA_N)`.

In the following, we show a simple example of PDDL *domain*.

Example 4.1. A simple PDDL *domain* the Tower of Hanoi game. This game consists of three rods and n disks of different size, which can slide into any rods. At the beginning, disks are arranged in a neat stack in ascending order of size on a rod, the smallest on the top. The goal of the game is to move the whole stack to another rod, following three rules:

- one disk at a time can be moved;

- a disk can be moved only if it is the uppermost disk on a stack;
- no disk can be placed on top of a smaller disk.

```
1 (define (domain hanoi) ;comment
2   (:requirements :strips :negative-preconditions :equality)
3   (:predicates (clear ?x) (on ?x ?y) (smaller ?x ?y) )
4   (:action move
5     :parameters (?disc ?from ?to)
6     :precondition (and
7        (smaller ?disc ?to) (smaller ?disc ?from)
8        (on ?disc ?from)
9        (clear ?disc) (clear ?to)
10       (not (= ?from ?to))
11    )
12    :effect (and
13      (clear ?from)
14      (on ?disc ?to)
15      (not (on ?disc ?from))
16      (not (clear ?to))
17    )
18  )
19 )
```

The PDDL *domain* file of the Tower of Hanoi is quite simple. Indeed, it consists of only one action (`move`) and only a few predicates. Firstly, the name given to this *domain* is `hanoi`. Then, there have been specified requirements as `:strips`, `:negative-preconditions` and `:equality`. After that, at line 3, there is the definition of all predicates involved in the PDDL *domain*. In particular, there are three predicates to describe if the top of a disk is `clear`, which disk is `on` top of another and, finally, which disk is `smaller` than another. Finally, there is the `move` action declaration with its parameters, its precondition formula and its effect formula.

The *problem* file

After having examined how a PDDL *domain* is defined, we can see the formulation of a PDDL *problem*. A PDDL *problem* is what a planner tries to solve. The *problem* file has the following format:

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
  )
```

At a first glance, we can notice that the *problem* definition includes the specification of the domain to which it is related. Indeed, every problem is defined with respect to a precise *domain*. Then, there is the object list which could be typed or untyped. After that, there are the initial and goal specification, respectively. The former defines what is true at the beginning of the planning task and it consists of ground atoms, namely predicates instantiated with previously defined objects. Finally, the goal description represents the formula, consisting of instantiated predicates, that we would like to achieve and obtain as a final state. In the following, we show a simple example of PDDL *problem*.

Example 4.2. In this example, we show a possible PDDL *problem* for the Tower of Hanoi game for which we have shown the *domain* in the Example 4.1.

```
1  (define (problem hanoi-prob)
2    (:domain hanoi)
3    (:objects rod1 rod2 rod3 d1 d2 d3)
4    (:init
5       (smaller d1 rod1) (smaller d2 rod1) (smaller d3 rod1)
6       (smaller d1 rod2) (smaller d2 rod2) (smaller d3 rod2)
7       (smaller d1 rod3) (smaller d2 rod3) (smaller d3 rod3)
8       (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
9       (clear rod2) (clear rod3) (clear d1)
10      (on d3 rod1) (on d2 d3) (on d1 d2))
11   (:goal (and (on d3 rod3) (on d2 d3) (on d1 d2)))
12 )
```

At line 3, we have three rods and three disks. At the beginning, all instantiated predicates that are true are mentioned. If a predicate is not mentioned, it is considered to be false. In the initial situation there have been specified all possible movements with the `smaller` predicate, the disks are one on top of the other in ascending order on `rod1` whereas the other two rods are `clear`. In addition, the goal description is a conjuctive formula requiring disks on a stack on the `rod3`.

Once both PDDL *domain* and a *problem* are specified, they are given as input to planners.

4.2.2 Fully Observable Non Deterministic Planning

In this Section, we formally define what *Fully Observable Non Deterministic Planning* (FOND) is giving some notions and definitions. Initially, we recall some concepts of "classical" planning while assuming the reader to be acquainted with basics of planning.

Given a PDDL specification with a *domain* and its corresponding *problem*, we would like to solve this specification in order to find a sequence of actions such that the goal formula holds true at the end of the execution. A *plan* is exactly that sequence of actions which leads the agent to achieve the goal starting from the initial state. Formally, we give the following definition.

Definition 4.1. A planning problem is defined as a tuple $\mathcal{P} = \langle \Sigma, s_0, g \rangle$, where:

- Σ is the state-transition system;
- s_0 is the initial state;
- g is the goal state.

Given the above Definition 4.1, we can formally define what a plan is.

Definition 4.2. A *plan* is any sequence of actions $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is a ground instance of an operator defined in the domain description.

Moreover, we have that:

Definition 4.3. A *plan* is a solution for $\mathcal{P} = \langle \Sigma, s_0, g \rangle$, if it is executable and achieves g.

Furthermore, a "classical" planning problem, just defined, is given under the assumptions of *fully observability* and *determinism*. In particular, the former means that the agent can always see the entire state of the environment whereas the latter means that the execution of an action is certain, namely any action that the agent takes uniquely determines its outcome.

Unlike the "classical" planning approach, in this thesis we focus on *Fully Observable Non Deterministic* (FOND) planning. Indeed, we continue relying on the *fully observability*, but loosing the *determinism*. In other words, in FOND planning we have the uncertainty on the outcome of an action execution. As anticipated in Section 4.2.1, the uncertainty of the outcome of an operator execution is syntactically expressed, in PDDL, with the keyword `oneof`. To better capture this concept, we give the following example.

Example 4.3. Here, we show as example the `put-on-block` operator of the FOND version of the well-known blocksworld PDDL *domain*.

```
(:action put-on-block
  :parameters (?b1 ?b2 - block)
  :precondition (and (holding ?b1) (clear ?b2))
  :effect (oneof (and (on ?b1 ?b2) (emptyhand) (clear ?b1)
                 (not (holding ?b1)) (not (clear ?b2)))
                 (and (on-table ?b1) (emptyhand) (clear ?b1)
                 (not (holding ?b1)))
          )
)
```

The effect of the `put-on-block` is non deterministic. Specifically, the action is executed every time the agent is holding a block and another block is clear on the top. The effect can be either that the block is put on top of the other block or that the block is put on table. This has to be intended as an aleatory event. Indeed, the agent does not control the operator execution result.

Additionally, a *non-deterministic* action a with effect $oneof(E_1, \dots, E_n)$ can be intended as a set of *deterministic* actions $b_1, \dots, b_n$, sharing the same precondition of a, but with effects $E_1, \dots, E_n$, respectively. Hence, the application of action a turns out in the application of one of the actions b_i, chosen non-deterministically.

At this point, we can formally define the FOND planning. Following (Ghallab et al., 2004) and (Geffner and Bonet, 2013), we give the following definition:

Definition 4.4. A *non-deterministic domain* is a tuple $\mathcal{D} = \langle 2^{\mathcal{F}}, A, s_0, \varrho, \alpha \rangle$ where:

- $\mathcal{F}$ is a set of *fluents* (atomic propositions);
- A is a set of *actions* (atomic symbols);
- $2^{\mathcal{F}}$ is the set of states;
- s_0 is the initial state (initial assignment to fluents);
- $\alpha(s) \subseteq A$ represents *action preconditions*;
- $(s, a, s') \in \varrho$ with $a \in \alpha(s)$ represents *action effects* (including frame assumptions).

Such domain $\mathcal{D}$ is assumed to be represented compactly (e.g. in PDDL), therefore, considering the *size* of the domain as the cardinality of $\mathcal{F}$. Intuitively, the evolution of a non-deterministic domain is as follows: from a given state s, the agent chooses what action $a \in \alpha(s)$ to execute, then, the environment chooses a *successor state* s' with $(s, a, s') \in \varrho$. To this extent, planning can also be seen as a *game* between two players: the agent tries to force eventually reaching the goal no matter how the environment behaves. Moreover, the agent can execute an action having the knowledge of all history of states so far.

Now, we can define the meaning of solving a FOND planning problem on $\mathcal{D}$. A *trace* of $\mathcal{D}$ is a finite or infinite sequence $s_0, a_0, s_1, a_1, \dots$ where s_0 is the initial state, $a_i \in \alpha(s_i)$ and $s_{i+1} \in \varrho(s_i, a_i)$ for each s_i, a_i in the trace.

Solutions to a FOND problem $\mathcal{P}$ are called *strategies* (or *policies*). A *strategy* π is defined as follows:

Definition 4.5. Given a FOND problem $\mathcal{P}$, a *strategy* π for $\mathcal{P}$ is a partial function defined as:

$$\pi : (2^{\mathcal{F}})^+ \rightarrow A \tag{4.1}$$

such that for every $u \in (2^{\mathcal{F}})^+$, if $\pi(u)$ is defined, then $\pi(u) \in \alpha(last(u))$, namely it selects applicable actions, whereas, if $\pi(u)$ is undefined, then $\pi(u) = \bot$.

A trace τ is *generated* by π (often called π-trace) if the following holds:

- if $s_0, a_0, \dots, s_i, a_i$ is a prefix of τ, then $\pi(s_0, s_1, \dots, s_i) = a_i$;
- if τ is finite, i.e. $\tau = s_0, a_0, \dots, a_{n-1}, s_n$, then $\pi(s_0, s_1, \dots, s_i) = \bot$.

For FOND planning problems, in (Cimatti et al., 2003), are defined different classes of solutions. Here we examine only two of them, namely *strong solution* and *strong cyclic solutions*. In the following, we give their formal definitions.

Definition 4.6. A *strong solution* is a strategy that is guaranteed to achieve the goal regardless of non-determinism.

Definition 4.7. *Strong cyclic solutions* guarantee goal reachability only under the assumption of *fairness*. In the presence of *fairness* it is supposed that all action outcomes, in a given state, would occur infinitely often.

Obviously, *strong cyclic solutions* are less restrictive than a *strong solution*. Indeed, as the name suggests, a strong cyclic solution may revisit states. However, in this thesis we will focus only on searching *strong solutions*. As final remark, when searching for a strong solution to a FOND problem we refer to FOND$_{sp}$.

In the next Section, we will generalize the concept of solving FOND planning problems with extended temporal goals, describing the step by step encoding process of those temporal goals in the FOND domain, written in PDDL.

4.3 The FOND4LTL$_f$/PLTL approach

As written in Section 4.1, planning with extended temporal goals has been considered over the representation of goals in classical planning to capture a richer class of plans where restrictions on the whole sequence of states must be satisfied as well. In particular, differently from classical planning, where the goal description can only be expressed as a propositional formula, in planning for extended temporal goals the goal description may have the same expressive power of the temporal logic in which the goal is specified. This enlarges the general view about planning. In other words, extended temporal goals specify desirable sequences of states and a plan exists if its execution yields one of these desirable sequences (Bacchus and Kabanza, 1998).

In this thesis, we propose a new approach, called FOND4LTL$_f$/PLTL, that uses LTL$_f$ and PLTL formalisms as temporal logics for expressing extended goals. To better understand the powerful of planning with extended temporal goals we give the following example.

Example 4.4. Considering the well known `triangle tireworld` FOND planning task. The objective is to drive from one location to another, however while driving a tire may be going flat. If there is a spare tire in the location of the car, then the car can use it to fix the flat tire. The task in depicted in Figure 4.1, where there are locations arranged as a triangle, arrows representing roads and circles meaning that in a location there is a spare tire.

A possible classical goal can be $G = vehicleAt(l31)$, namely a propositional formula saying something only about what have to be true at the end of the execution. In the case of G, we exclusively require that the vehicle should be in location $l31$.

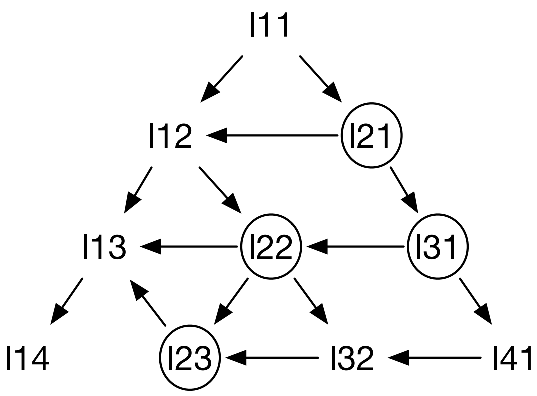


Figure 4.1. A possible Triangle Tireworld task. Locations marked with a circle have a spare tire, arrows represent possible directions

On the contrary, a goal specification expressed with temporal formalism such as PLTL could be $\varphi = vehicleAt(l13) \wedge \Diamond\!\!\!\!-(vehicleAt(l23))$. Such a specification requires to reach position $l13$, imposing the passage through position $l23$ before reaching the goal.

Planning for LTL$_f$ and PLTL goals slightly changes the definitions given in Section 4.2.2. In the following, we give the modified definitions of the concepts seen before.

Definition 4.8. Given a domain $\mathcal{D}$ and an LTL$_f$/PLTL formula φ over atoms $\mathcal{F} \cup \mathcal{A}$, a strategy π is a *strong solution* to $\mathcal{D}$ for goal φ, if every π-trace is finite and satisfies φ.

About complexity of FOND$_{sp}$, we have the following Theorems.

Theorem 4.1. *(De Giacomo and Rubin, 2018) Solving* FOND$_{sp}$ *for* LTL$_f$ *goals is:*

- EXPTIME-*complete in the size of the domain;*
- 2EXPTIME-*complete in the size of the goal.*

Furthermore, if the goal has the form of $\Diamond G$, i.e. is a reachability goal, the cost with respect to the goal becomes polynomial because it is just a propositional evaluation. If for a given LTL$_f$ goal the determinization step does not cause a state explosion, the complexity with respect to the goal is EXPTIME. On the contrary, if G is a PLTL formula, from results in De Giacomo and Rubin (2018), we can only say that the complexity is 2EXPTIME in the size of the goal. Even though we remark that the investigation on planning for PLTL goals may have a computational advantage since PLTL formulas can be reduced to DFA in single exponential time (vs. double-exponential time of LTL$_f$ formulas) (Chandra et al., 1981), computing the hardness is not obvious because we should evaluate the formula only after the $\Diamond$ operator.

4.3.1 Idea

Our FOND4LTL$_f$/PLTL approach works as follows: given a non-deterministic planning domain $\mathcal{D}$, an initial state s_0 and an LTL$_f$ or PLTL goal formula φ (whose symbols are ground predicates), we first obtain the corresponding DFA of the temporal formula through LTL$_f$2DFA, then, we encode such a DFA into the non-deterministic planning domain $\mathcal{D}$. As a result, we will have a new domain $\mathcal{D}'$ and a new problem P' that can be considered and solved as a classical FOND planning problem.

The new approach, carried out in this thesis, stems from the research in De Giacomo and Rubin (2018), that, basically, proposes automata-theoretic foundations of FOND planning for LTL$_f$ goals. In particular, they compute the cartesian product between the DFA corresponding to the domain $\mathcal{D}$ ($\mathcal{A}_\mathcal{D}$) and the DFA corresponding to φ ($\mathcal{A}_\varphi$), thus, solving a DFA game on $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$, i.e. find, if exists, a trace accepted by $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$. Moreover, an important consideration is that the resulting automaton $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$ will read both the action and its effect, as shown, for instance, in Figure 4.2.

However, unlike what has been done in De Giacomo and Rubin (2018), we split transitions containing the action and its effect in order to have them separately. The reason for this separation is that having both the action and its effect on the same transition is not suitable on a practical perspective. Hence, we have devised a solution in which we run $\mathcal{A}_\mathcal{D}$ and $\mathcal{A}_\varphi$ separately, but combining them into a single unique transition system. To achieve this, we move $\mathcal{A}_\mathcal{D}$ and $\mathcal{A}_\varphi$ alternatively by introducing an additional predicate, which we will call `turnDomain`, that is true when we should move $\mathcal{A}_\mathcal{D}$ and is false when we should move $\mathcal{A}_\varphi$. In the following, we give an example to better understand the solution put in place in this thesis.

Example 4.5. Let us consider the simplified version of the classical Yale shooting domain (Hanks and McDermott, 1986) as in De Giacomo and Rubin (2018), where we have that the turkey is either alive or not and the actions are shoot and wait with the obvious effects, but with a gun that can be faulty. Specifically, shooting with a (supposedly) working gun can either end in killing the turkey or in the turkey staying alive and the discovery that the gun is not working properly. On the other hand, shooting (with care) with a gun that does not work properly makes it work and kills the turkey. The cartesian product $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$ with $\varphi = \Diamond\neg a$ is as follows:

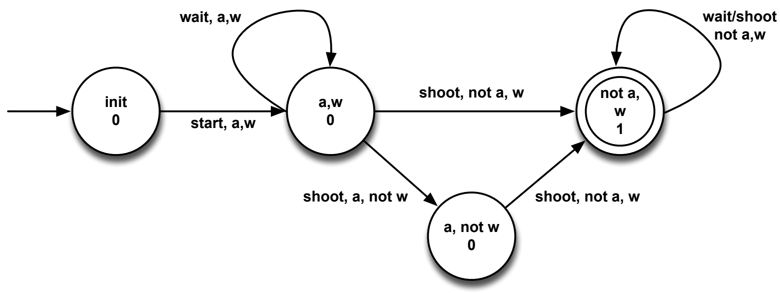


Figure 4.2. The DFA corresponding to $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$. Symbol `a` stands for *alive* and `w` for *working*

As we can see in Figure 4.2, each transition reads both the action and its effect. This is not suitable for a practical implementation. Thus, we do not perform the cartesian product between the two automata. On the contrast, we build a transition system as in Figure 4.3.

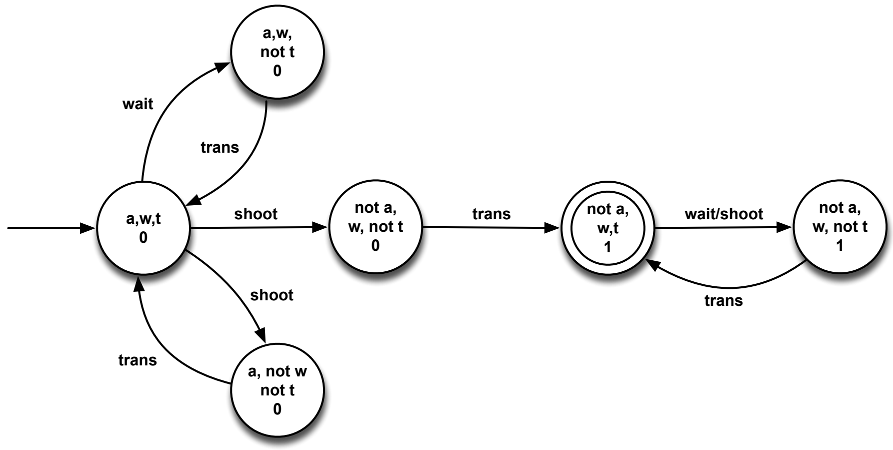


Figure 4.3. The new transition system corresponding to the Yale shooting domain. Symbol `a` stands for *alive*, `w` for *working* and `t` for *turnDomain*

The transition system, shown in Figure 4.3, expresses the new domain $\mathcal{D}'$ that has a perfect alternation of transitions. In particular, actions of the initial domain $\mathcal{D}$ alternates with a special action, that we called `trans`, representing the movement done by $\mathcal{A}_\varphi$. Moreover, it is important to notice the usage of the added predicate `turnDomain` allowing us to alternate between general actions and the new action `trans`.

In the next Section, we will explain how the new domain $\mathcal{D}'$ and the new problem $\mathcal{P}'$ can be written in PDDL by showing the encoding of LTL$_f$/PLTL goals in PDDL.

4.3.2 Encoding of LTL$_f$/PLTL goals in PDDL

In this Section, we describe the process of obtaining the new domain $\mathcal{D}'$ and the new problem $\mathcal{P}'$, both specified in PDDL. The original PDDL domain $\mathcal{D}$ and the associated original problem $\mathcal{P}$ change when introducing LTL$_f$/PLTL goals. In particular, what changes is the way we encode our LTL$_f$/PLTL formula in PDDL. Firstly, we employ our LTL$_f$2DFA tool to convert the given goal formula φ into the corresponding DFA. Then, we encode, in a specific way, the resulting DFA automaton in PDDL modifying the original domain $\mathcal{D}$ and problem $\mathcal{P}$.

Translation of DFAs to PDDL in Domain $\mathcal{D}$

In order to explain the translation technique of a DFA to PDDL, we assume to already have the DFA generated by LTL$_f$2DFA. We recall that such a DFA is formally defined as follows:

Definition 4.9. A DFA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where:

- $\Sigma = \{a_0, a_1, \ldots, a_n\}$ is a finite set of symbols;
- $Q = \{q_0, q_1, \ldots, q_m\}$ is the finite set of states;
- $q_0 \in Q$ is the initial state;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $F \subseteq Q$ is the set of final states;

Specifically, since the automaton $\mathcal{A}$ corresponds to the goal formula φ, which has grounded predicates as symbols, we can represent them as $\{a_0(o_0, \ldots, o_j), \ldots, a_n(o_0, \ldots, o_w)\}$, where $o_0, \ldots, o_k \in \mathcal{O}$ and $0 \leq j, w \leq k$ represents objects of the domain $\mathcal{D}$. To capture the general representation of φ in $\mathcal{D}$, we have to modify $\mathcal{A}$ to $\mathcal{A}'$ performing a transformation explained below. We give the following definitions.

Definition 4.10. $\mathcal{A}'$ is a tuple $\mathcal{A}' = \langle \Sigma', Q', q_0', \delta', F' \rangle$, where:

- $\Sigma' = \{a_0', a_1', \ldots, a_n'\}$ is a finite set of symbols;
- $Q' = \{q_0', q_1', \ldots, q_m'\}$ is the finite set of states;
- $q_0' \in Q$ is the initial state;
- $\delta' : Q' \times \Sigma' \to Q'$ is the transition function;
- $F' \subseteq Q'$ is the set of final states;

Definition 4.11. Given the set of DFA symbols Σ, we define a mapping function f as follows:

$$f : \mathcal{O} \to \mathcal{V} \tag{4.2}$$

where $\mathcal{O}$ is the set of objects $\{o_0, \ldots, o_k\}$ and $\mathcal{V}$ is a set of variables $\{x_0, \ldots, x_k\}$

The transformation from $\mathcal{A}$ and $\mathcal{A}'$ is carried out with the mapping function f as follows:

- $\Sigma' = \{a_0', \ldots, a_n'\}$, where $a_i' \doteq a_i(x_0, \ldots, x_j)$ and $x_0, \ldots, x_j \subseteq \mathcal{V}$;
- $Q' = \{q_0', \ldots, q_m'\}$, where $q_i' \doteq q_i(x_0 \ldots, x_k)$.

Then, q_0', δ' and F' are modified accordingly.

Once the transformation is done, we have obtained a parametric DFA which is a general representation with respect to the original one. After that, for representing the DFA transitions in the domain $\mathcal{D}$, we should encode the new *transition function* δ' into PDDL. To this extent, the δ' function is represented as a new PDDL operator, called `trans` having these properties:

- all variables in $\mathcal{V}$ are parameters;

- the negation of the `turnDomain` predicate is a precondition;
- effects represent the δ' function.

Moreover, effects are expressed as conditional effects. The general encoding would be as follows:

Action **trans**:
parameters: $(x_0, \ldots, x_k)$, where $x_i \in \mathcal{V}$
preconditions: $\neg turnDomain$
effects: when $(q_i(x_0, \ldots, x_k) \wedge a'_j)$ then $(\delta'(q'_i, a'_j) = q''_i(x_0, \ldots, x_k) \wedge (\neg q, \forall q \in Q$ s.t. $q \neq q''_i) \wedge turnDomain)$, $\forall i, j : 0 \leq i \leq m, 0 \leq j \leq n$

Additionally, in PDDL, especially in the effect formula of a conditional effect, we should specify that if the automaton is in a state, it is not in other states. This is captured by adding the negation of all other automaton states. In the following, we give an example showing the translation of DFAs to PDDL step-by-step.

Example 4.6. Let us consider the goal formula $\varphi = \Diamond(on(d3, rod3))$ for the Tower of Hanoi planning problem. The predicate `on` is instantiated on objects `d3` and `rod3`. By applying the mapping function f we have the corresponding variables and φ becomes $\varphi(x_1, x_2) = \Diamond(on(x_1, x_2))$, where we know that $x_1 = f(d_3)$ and $x_2 = f(rod_3)$. In this case, the modified DFA $\mathcal{A}'$ is depicted in Figure 4.4. At this point, consider the new DFA,

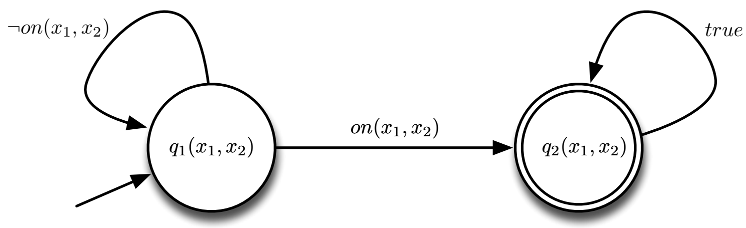


Figure 4.4. The parametric DFA corresponding to $\varphi(x_1, x_2) = \Diamond(on(x_1, x_2))$

the `trans` operator built from that automaton is the following:

```
(:action trans
  :parameters (?x1 ?x2)
  :precondition (not (turnDomain))
  :effect (and (when (and (q1 ?x1 ?x2) (not (on ?x1 ?x2)))
                  (and (q1 ?x1 ?x2) (not (q2 ?x1 ?x2)) (turnDomain))
              (when (or (and (q1 ?x1 ?x2) (on ?x1 ?x2)) (q2 ?x1 ?x2))
                  (and (q2 ?x1 ?x2) (not (q1 ?x1 ?x2)) (turnDomain))
  )
)
```

As just shown in Example 4.6, transitions, with source state and destination state, are encoded as conditional effects, where the condition formula includes source state and formula symbols whereas the effect formula includes the destination state, the negation of all other states and `turnDomain`. Moreover, in order to get a compact encoding of `trans` effects, conditional effects are brought together by destination state as happens, for instance, at line 6.

After the `trans` operator has been built, we change $\mathcal{D}$ as follows:

1. $\forall a \in A$: add `turnDomain` to $\alpha(s)$, i.e add `turnDomain` predicate to all actions precondition $\alpha(s)$;
2. $\forall a \in A$: add `(not (turnDomain))` to $(s, a, s') \in \varrho$ with $a \in \alpha(s)$, i.e add negated `turnDomain` to all actions effects (s, a, s');
3. add `trans` operator;
4. $\forall q' \in Q'$: add q' to predicates definition of $\mathcal{D}$, i.e. add all automaton state predicates to the domain predicates definition.

We have thus obtained the new domain $\mathcal{D}'$. In the following, we show an example.

Example 4.7. Let consider again the Triangle Tireworld scenario. The original PDDL domain is:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location ?to - location)
               (not-flattire))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicle-at ?from) (road ?from ?to)
      (not-flattire))
    :effect (oneof (and (vehicle-at ?to) (not (vehicle-at ?from)))
          (and (vehicle-at ?to) (not (vehicle-at ?from))
          (not (not-flattire))))
   )
  (:action changetire
    :parameters (?loc - location)
    :precondition (and (spare-in ?loc) (vehicle-at ?loc))
    :effect (and (not (spare-in ?loc)) (not-flattire))
   )
)
```

Now, consider a simple LTL$_f$ formula $\varphi = \Diamond vehicleAt(l13)$. It requires that eventually the vehicle will be in location $l13$. The parametric DFA associated to $\varphi(x)$ is depicted in Figure 4.5.

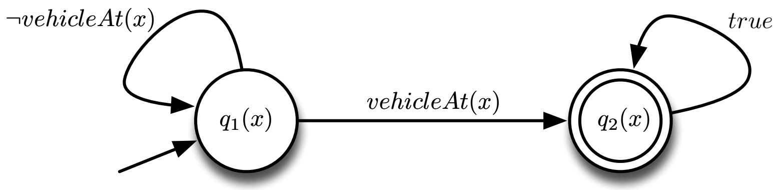


Figure 4.5. The parametric DFA corresponding to $\varphi(x) = \Diamond(vehicleAt(x))$

Considering the DFA in Figure 4.5, the **trans** operator built from that automaton is the following:

```
(:action trans
  :parameters (?x - location)
  :precondition (not (turnDomain))
  :effect (and (when (and (q1 ?x) (not (vehicle-at ?x)))
               (and (q1 ?x) (not (q2 ?x)) (turnDomain))
           (when (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
               (and (q2 ?x) (not (q1 ?x)) (turnDomain))
  )
)
```

Finally, putting together all pieces and carrying out changes described above, we obtain the new domain $\mathcal{D}'$ as follows:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location ?to - location)
               (not-flattire)
               (q1 ?x - location)
               (q2 ?x - location)
               (turnDomain))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicle-at ?from) (road ?from ?to)
      (not-flattire) (turnDomain))
    :effect (oneof (and (vehicle-at ?to) (not (vehicle-at ?from))
      (not (turnDomain)))
          (and (vehicle-at ?to) (not (vehicle-at ?from))
          (not (not-flattire)) (not (turnDomain))))
   )
  (:action changetire
    :parameters (?loc - location)
```

```
    :precondition (and (spare-in ?loc) (vehicle-at ?loc)
      (turnDomain))
    :effect (and (not (spare-in ?loc)) (not-flattire)
      (not (turnDomain)))
   )
  (:action trans
    :parameters (?x - location)
    :precondition (not (turnDomain))
    :effect (and (when (and (q1 ?x) (not (vehicle-at ?x)))
              (and (q1 ?x) (not (q2 ?x)) (turnDomain))
          (when (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
              (and (q2 ?x) (not (q1 ?x)) (turnDomain)))
  )
)
```

Change in Problem $\mathcal{P}$

Concerning the planning problem $\mathcal{P}$, we completely discard the goal specification, whereas the initial state description is slightly modified. Moreover, the problem name, the associated domain name and all defined objects remain unchanged. We have to modify both the initial state and the goal state specifications to make them compliant with $\mathcal{D}'$, containing all changes introduced in the planning domain $\mathcal{D}$. To this extent, we formally define the new initial state as follows:

$$\text{Init: } s_0 \wedge \texttt{turnDomain} \wedge \hat{q}'_0 \tag{4.3}$$

where $\hat{q}'_0 \doteq q_0(f^{-1}(x_0), \dots, f^{-1}(x_k)) = q_0(o_0, \dots, o_k)$. In other words, we put together the original initial specification s_0, the new predicate `turnDomain`, meaning that it is *true* at the beginning, and the initial state of the automaton instantiated on the objects of interest, i.e. those specified in the LTL$_f$/PLTL formula φ.

On the other hand, the goal description is built from scratch as follows:

$$\text{Goal: } \texttt{turnDomain} \wedge \Big(\bigvee_{\hat{q}' \in F} \hat{q}' \Big) \tag{4.4}$$

where $\hat{q}' \doteq q(f^{-1}(x_0), \dots, f^{-1}(x_k)) = q(o_0, \dots, o_k)$. In other words, we place together the `turnDomain` predicate, meaning that it must be *true* at the end of the execution, and the final state(s) of the automaton always instantiated on the objects of interest. Here, it is important to notice that if the automaton has two or more final states, they should be put in disjunction.

We can give the following example.

Example 4.8. Let consider again the Triangle Tireworld scenario, shown in the Example 4.7. The original PDDL domain is:

```
(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
  (:init (vehicle-at l11)
    (road l11 l12) (road l12 l13) (road l11 l21) (road l12 l22)
    (road l21 l12) (road l22 l13) (road l21 l31) (road l31 l22)
    (spare-in l21) (spare-in l22) (spare-in l31)
    (not-flattire))
  (:goal (vehicle-at l13))
)
```

Now, considering the same LTL$_f$ formula $\varphi = \Diamond vehicleAt(l13)$, the object of interest is $l13$. Hence, we should evaluate our automaton states as $q_1(l13)$ and $q_2(l13)$.

Finally, putting together all pieces and carrying out changes described above, we obtain the new problem $\mathcal{P}'$ as follows:

```
(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
  (:init (vehicle-at l11)
    (road l11 l12) (road l12 l13) (road l11 l21) (road l12 l22)
    (road l21 l12) (road l22 l13) (road l21 l31) (road l31 l22)
    (spare-in l21) (spare-in l22) (spare-in l31)
    (not-flattire) (turnDomain) (q1 l13))
  (:goal (and (turnDomain) (q2 l13)))
)
```

As a remark, we will refer to the new goal specification as $\mathcal{G}'$.

Having examined the encoding of LTL$_f$/PLTL goal formulas in PDDL, the resulting planning domain $\mathcal{D}'$ and problem $\mathcal{P}'$ represent a "classical" planning specification. In the next Section, we will see how we obtain a strong policy giving $\mathcal{D}'$ and $\mathcal{P}'$.

4.3.3 FOND Planners

In this Section, we talk about the state-of-art FOND planners and how they are employed within this thesis.

To begin with, thanks to our encoding process, we have reduced the problem of FOND planning for LTL$_f$/PLTL goals to a "classical" FOND planning, which is essentially a *reachability* problem. We can state the following Theorem.

Theorem 4.2. *A strong policy π is a valid policy for $\mathcal{D}', \mathcal{G}'$ if and only if π is a valid policy for $\mathcal{D}, \varphi$.*

Given this Theorem, we can solve our original problem giving $\mathcal{D}'$ and $\mathcal{P}'$ as input to standard FOND planners. The main state-of-art FOND planners are:

- MBP and Gamer, which are OBDD[2]-based planners (Cimatti et al., 2003; Kissmann and Edelkamp, 2009)
- MyND and Grendel, which rely on explicit AND/OR graph search (Bercher, 2010; Ramirez and Sardina, 2014)
- PRP, NDP and FIP, which rely on classical algorithms (Kuter et al., 2008; Fu et al., 2011; Muise et al., 2012)
- FOND-SAT, which provides a SAT approach to FOND planning (Geffner and Geffner, 2018)

Although FOND planning is receiving an increasingly interest, the research on computational approaches has been recently reduced. Nevertheless, some planners performs well on different contexts of use. In our thesis, we are going to employ a customized version of FOND-SAT, the newest planner.

Secondly, although the description of many real-world planning problems involves the use of conditional effects, requiring full support of ADL by planners, those state-of-art planners, cited above, are still not able to fully handle such conditional effects. This represents a big limitation that can be surely deepened as a future work of this thesis. To this extent, we should first compile away conditional effects from the domain and, then, we can give it to a planner. Our proposal implementation, thoroughly described in the Section 4.4, is able to compile away simple conditional effects, namely those conditional effects that do not have nested formulas. Additionally, we have to compile away conditional effects of the `trans` operator upstream even though its representation with the employment of conditional effects is much more effective and compact. Luckily, this process consists of just splitting the operator in as many operators as the number of conditional effects present in the original action and adding the condition formula in the preconditions for each conditional effect. In the following, we make a clarifying example.

Example 4.9. The `trans` operator built in the Example 4.7 is:

```
(:action trans
  :parameters (?x - location)
  :precondition (not (turnDomain))
  :effect (and (when (and (q1 ?x) (not (vehicle-at ?x)))
              (and (q1 ?x) (not (q2 ?x)) (turnDomain))
          (when (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
              (and (q2 ?x) (not (q1 ?x)) (turnDomain))
  )
)
```

As we can see, it contains only two conditional effects. Hence, we split this operator in two operators that we are going to call `trans-0` and `trans-1`, respectively. In particular, for each conditional effect the condition formula is added to the precondition and the effect formula is left in the effects. `trans-0` and `trans-1` are as follows:

[2] OBDD stands for *Ordered Binary Decision Diagram*

```
(:action trans-0
  :parameters (?x - location)
  :precondition (and (and (q1 ?x) (not (vehicle-at ?x)))
    (not (turnDomain)))
  :effect (and (q1 ?x) (not (q2 ?x)) (turnDomain))
  )
)
(:action trans-1
  :parameters (?x - location)
  :precondition (and (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
    (not (turnDomain)))
  :effect (and (q2 ?x) (not (q1 ?x)) (turnDomain))
  )
)
```

At this point, having compiled away simple conditional effects from the modified domain $\mathcal{D}'$, we can finally describe how we have employed FOND-SAT in our thesis.

The main reasons why we have chosen FOND-SAT are the following:

- it is written in pure Python, hence we can easily integrate it in our Python implementation as we will see in Section 4.4;
- it performs reasonably well;
- it outputs all policies building a transition system whose states are called *controller states*.

FOND-SAT takes also advantage of the parser and translation PDDL-to-SAS+ scripts from PRP. When FOND-SAT was developed, PRP's translation scripts could not handle disjunctive preconditions that may be present in our `trans` operators. As a result, we have modified FOND-SAT with the newest version of those scripts directly from PRP.

The usage of FOND-SAT is really simple. From its source folder, it is necessary to run the following command in the terminal:

```
python main.py -strong 1 -policy 1 /path-to/domain.pddl
/path-to/problem.pddl
```

The command simply executes the main module of FOND-SAT requiring to find strong policies and to print, if exists, the policy found. We feed FOND-SAT with our new domain $\mathcal{D}'$ and new problem $\mathcal{P}'$.

Once strong plans are found, FOND-SAT displays the policy in four sections as follows:

- Atom (CS): for each controller state it tells what predicates are true;
- (CS, Action with arguments): for each controller state it tells which actions can be applied

- (CS, Action name, CS): it tells for each controller state what action is applied in that state and the successor state;
- (CS1, CS2): it means that the controller can go from CS1 to CS2.

Now, we give an output example.

Example 4.10. The following result has been obtained running FOND-SAT with the Triangle Tireworld domain and problem with the LTL$_f$ goal $\varphi = \Diamond vehicleAt(l31)$. What follows is only the displayed policy.

```
...
Trying with 7 states
Looking for strong plans: True
Fair actions: True
# Atoms: 18
# Actions: 26
SAT formula generation time = 0.052484
# Clauses = 11041
# Variables = 1225
Creating formula...
Done creating formula. Calling solver...
SAT solver called with 4096 MB and 3599 seconds
Done solver. Round time: 0.016456
Cumulated solver time: 0.055322
==========================
==========================
Controller -- CS = Controller State
==========================
==========================
Atom (CS)
__________________________
--------------
Atom q1(l31) (n0)
Atom vehicleat(l11) (n0)
Atom not-flattire() (n0)
Atom spare-in(l21) (n0)
Atom turndomain() (n0)
--------------
-NegatedAtom turndomain() (n1)
Atom q1(l31) (n1)
Atom vehicleat(l21) (n1)
Atom not-flattire() (n1)
--------------
Atom q1(l31) (n2)
```

```
-NegatedAtom turndomain() (n2)
Atom spare-in(l21) (n2)
Atom vehicleat(l21) (n2)
----------
Atom turndomain() (n3)
Atom q1(l31) (n3)
Atom vehicleat(l21) (n3)
Atom not-flattire() (n3)
----------
Atom q1(l31) (n4)
Atom spare-in(l21) (n4)
Atom vehicleat(l21) (n4)
Atom turndomain() (n4)
----------
Atom q1(l31) (n5)
Atom vehicleat(l31) (n5)
-NegatedAtom turndomain() (n5)
----------
Atom turndomain() (ng)
Atom q2(l31) (ng)
====================
====================
(CS, Action with arguments)
____________________
(n0,move-car_DETDUP_0(l11,l21))
(n0,move-car_DETDUP_1)
(n0,move-car_DETDUP_1(l11,l21))
(n0,move-car_DETDUP_0)
(n1,trans-0_v4)
(n1,trans-0_v4(l31))
(n2,trans-0_v4(l31))
(n2,trans-0_v4)
(n3,move-car_DETDUP_0(l21,l31))
(n3,move-car_DETDUP_0)
(n3,move-car_DETDUP_1(l21,l31))
(n3,move-car_DETDUP_1)
(n4,changetire(l21))
(n4,changetire)
(n5,trans-11)
(n5,trans-11(l31))
====================
====================
(CS, Action name, CS)
```

```
________________________
(n0,move-car_DETDUP_0,n1)
(n0,move-car_DETDUP_1,n2)
(n1,trans-0_v4,n3)
(n2,trans-0_v4,n4)
(n3,move-car_DETDUP_0,n5)
(n3,move-car_DETDUP_1,n5)
(n4,changetire,n1)
(n5,trans-11,ng)
========================
(CS, CS)
________________________
(n2,n4)
(n5,ng)
(n3,n5)
(n1,n3)
(n0,n2)
(n4,n1)
(n0,n1)
========================
Solved with 7 states
Elapsed total time (s): 0.288035
Elapsed solver time (s): 0.055322
Elapsed solver time (s): [0.0052, 0.0059, 0.007, 0.009, 0.011, 0.016]
Looking for strong plans: True
Fair actions: True
Done
```

As we can see, operators names are changed due to internal arrangements made by FOND-SAT needed to handle both non determinism and disjunctive preconditions. Here, it is important to observe that, as we expected, there is an alternation of action executions between the original domain actions and the `trans` operators. Finally, the transition system built by FOND-SAT has $n0$ as initial state and ng as final state. If strong plans are found, it means that every path from $n0$ to ng is a valid plan. We will better explain this later in Section 4.5.

In the following sections, we will describe in details our practical implementation, called FOND4LTL$_f$/PLTL, that automates all the process illustrated in this Section.

4.4 Implementation

In this section, we thoroughly describe the proposed implementation of concepts given in Section 4.3. In particular, we give some general information about its features, dependencies and usage. Then, we focus on the package explaining how is structured and commenting highlights on the code.

We decided to call the package FOND4LTL$_f$/PLTL, enhancing the possibility to solve FOND planning also for PLTL goals, which is a novelty in this area of research and application. Moreover, the package has been developed in pure Python and has the following main features:

- perform FOND planning for LTL$_f$ or PLTL goals;
- compiling simple ADL conditional effects from the planning domain;
- encode LTL$_f$/PLTL formulas into standard PDDL (McDermott et al., 1998).

These features are achieved together with the integration of the LTL$_f$2DFA tool (Chapter 3).

Secondly, FOND4LTL$_f$/PLTL requires Python>=3.6 and has the following dependencies:

- LTL$_f$2DFA, presented in Chapter 3. It has been used for the generation of DFAs corresponding to LTL$_f$/PLTL goal formulas;
- PLY, a pure-Python implementation of the popular compiler construction tools Lex and Yacc. It has been employed for PDDL parsing;
- pydot, a Python interface to GraphViz format language. It has been employed for DFAs parsing;
- possible other dependencies linked to the employed planner. In the case of FOND-SAT, it has MINISAT as dependency.

The FOND4LTL$_f$/PLTL software is an open-source project and available to download on GitHub[3].

Additionally, FOND4LTL$_f$/PLTL will soon be available as an online tool at the website address http://fond4ltlfpltl.diag.uniroma1.it.

4.4.1 Package Structure

The structure of the FOND4LTL$_f$/PLTL package is ordered and divided according to the scope of each single module. It consists of the following:

- `pddl/`: it is the directory containing all the necessary code for parsing the PDDL standard and all data structures designed and implemented to handle PDDL;
- `automa/`: it consists of the `automa.py` file, the `aparser.py` file and the `symbol.py` file. In this folder, we find all the code for dealing with automata;
- `fond4ltlfpltl.py`: it is the main module of the package.

[3]https://github.com/Francesco17/FOND4LTLfPLTL

4.4.2 PDDL

In this section, we illustrate the code that handles the PDDL standard. More specifically, we talk about the parsing of PDDL domains and problems showing all implemented data structures.

First of all, thanks to the PLY library we have implemented the `lexer.py` and the `parser.py` modules enabling the parsing of a PDDL document. For a brief overview of the functioning of PLY we refer the reader to Sections 3.2.1 and 3.2.2. Indeed, the operation is exactly the same. In the following, we show the listing of the two modules focusing only on the most important parts. Three dots ... represent omitted code.

Listing 4.1. The `MyLexer` class.

```
import ply.lex as lex

class MyLexer(object):

    reserved = {
        'define':                    'DEFINE_KEY',
        'domain':                    'DOMAIN_KEY',
        ':domain':                   'DOMAIN_PKEY',
        ':requirements':             'REQUIREMENTS_KEY',
        ':constants':                'CONSTANTS_KEY',
        ':strips':                   'STRIPS_KEY',
        ':adl':                      'ADL_KEY',
        ':non-deterministic':        'ND_KEY',
        ':equality':                 'EQUALITY_KEY',
        ':typing':                   'TYPING_KEY',
        ':types':                    'TYPES_KEY',
        ':predicates':               'PREDICATES_KEY',
        ':action':                   'ACTION_KEY',
        ':parameters':               'PARAMETERS_KEY',
        ':precondition':             'PRECONDITION_KEY',
        ':effect':                   'EFFECT_KEY',
        'and':                       'AND_KEY',
        'or':                        'OR_KEY',
        'not':                       'NOT_KEY',
        'imply':                     'IMPLY_KEY',
        'oneof':                     'ONEOF_KEY',
        'forall':                    'FORALL_KEY',
        'exists':                    'EXISTS_KEY',
        'when':                      'WHEN_KEY',
        'problem':                   'PROBLEM_KEY',
        ':objects':                  'OBJECTS_KEY',
        ':init':                     'INIT_KEY',
```

```
        ':goal':                    'GOAL_KEY'
    }

    tokens = (
         'NAME',
         'VARIABLE',
         'LPAREN',
         'RPAREN',
         'HYPHEN',
         'EQUALS'
    ) + tuple(reserved.values())

    t_LPAREN = r'\('
    t_RPAREN = r'\)'
    t_HYPHEN = r'\-'
    t_EQUALS = r'='

    t_ignore = ' \t'

    def t_KEYWORD(self, t):
        r':?[a-zA-z_][a-zA-Z_0-9\-]*'
        t.type = self.reserved.get(t.value,'NAME')
        return t

    def t_NAME(self, t):
        r'[0-9a-zA-z_][a-zA-Z_0-9\-]*'
        return t

    def t_VARIABLE(self, t):
        r'\?[a-zA-z_][a-zA-Z_0-9\-]*'
        return t

    def t_COMMENT(self, t):
        r';.*'
        pass
...
    def t_newline(self, t):
        r'\n+'
        t.lineno += len(t.value)
...
```

As we can see, we have defined all tokens present in PDDL plus the *oneof* keyword, which is used to parse also non deterministic actions. Then, we have specified all regular expression rules for those tokens.

On the other hand, following the official PDDL standard grammar syntax, we have implemented the `MyParser` class as follows:

Listing 4.2. The `MyParser` class.

```
...
class MyParser(object):
...

    def p_pddl(self, p):
        '''pddl : domain
                | problem'''
        p[0] = p[1]

    def p_domain(self, p):
        '''domain : LPAREN DEFINE_KEY domain_def require_def types_def
        constants_def predicates_def action_def_lst RPAREN
                  | LPAREN DEFINE_KEY domain_def require_def types_def
                  predicates_def action_def_lst RPAREN
                  | LPAREN DEFINE_KEY domain_def require_def predicates_def
                  action_def_lst RPAREN'''
        if len(p) == 10:
            p[0] = Domain(p[3], p[4], p[5], p[6], p[7], p[8])
        elif len(p) == 9:
            p[0] = Domain(p[3], p[4], p[5], [], p[6], p[7])
        else:
            p[0] = Domain(p[3], p[4], [], [], p[5], p[6])

    def p_problem(self, p):
        '''problem : LPAREN DEFINE_KEY problem_def domain_pdef objects_def
        init_def goal_def RPAREN'''
        p[0] = Problem(p[3], p[4], p[5], p[6], p[7])
...
    def p_init_def(self, p):
        '''init_def : LPAREN INIT_KEY LPAREN AND_KEY ground_predicates_lst
        RPAREN RPAREN
                    | LPAREN INIT_KEY ground_predicates_lst RPAREN'''
        if len(p) == 5:
            p[0] = p[3]
        elif len(p) == 8:
            p[0] = p[5]

    def p_goal_def(self, p):
        '''goal_def : LPAREN GOAL_KEY LPAREN AND_KEY ground_predicates_lst
        RPAREN RPAREN
```

```
                    | LPAREN GOAL_KEY ground_predicates_lst RPAREN'''
        if len(p) == 7:
            p[0] = p[5]
        else:
            p[0] = p[3]
...
    def p_predicates_def(self, p):
        '''predicates_def : LPAREN PREDICATES_KEY predicate_def_lst RPAREN
        '''
        p[0] = p[3]

    def p_predicate_def(self, p):
        '''predicate_def : LPAREN NAME typed_variables_lst RPAREN
                         | LPAREN NAME RPAREN'''
        if len(p) == 4:
            p[0] = Predicate(p[2])
        elif len(p) == 5:
            p[0] = Predicate(p[2], p[3])
...
    def p_action_def(self, p):
        '''action_def : LPAREN ACTION_KEY NAME parameters_def precond_def
        effects_def RPAREN'''
        p[0] = Action(p[3], p[4], p[5], p[6])

    def p_parameters_def(self, p):
        '''parameters_def : PARAMETERS_KEY LPAREN typed_variables_lst RPAREN
                          | PARAMETERS_KEY LPAREN RPAREN'''
        if len(p) == 4:
            p[0] = []
        elif len(p) == 5:
            p[0] = p[3]

    def p_precond_def(self, p):
        '''precond_def : PRECONDITION_KEY LPAREN formula RPAREN'''
        p[0] = p[3]

    def p_formula(self, p):
        '''formula : literal
                   | AND_KEY formula_lst
                   | OR_KEY formula_lst
                   | NOT_KEY formula
                   | IMPLY_KEY formula formula
                   | EXISTS_KEY LPAREN typed_variables_lst RPAREN formula
```

```
                    | FORALL_KEY LPAREN typed_variables_lst RPAREN formula
                    | LPAREN AND_KEY formula_lst RPAREN
                    | LPAREN OR_KEY formula_lst RPAREN
                    | LPAREN NOT_KEY formula RPAREN
                    | LPAREN IMPLY_KEY formula formula RPAREN
                    | LPAREN literal RPAREN
                    | LPAREN EXISTS_KEY LPAREN typed_variables_lst RPAREN
                    formula RPAREN
                    | LPAREN FORALL_KEY LPAREN typed_variables_lst RPAREN
                    formula RPAREN'''
        if len(p) == 2:
            p[0] = p[1]
        elif len(p) == 3:
            if p[1] == 'and':
                p[0] = FormulaAnd(p[2])
            elif p[1] == 'or':
                p[0] = FormulaOr(p[2])
            elif p[1] == 'not':
                p[0] = FormulaNot(p[2])
        elif len(p) == 4:
            if p[1] == 'imply':
                p[0] = FormulaImply(p[2], p[3])
            else:
                p[0] = p[2]
        elif len(p) == 5:
            if p[2] == 'and':
                p[0] = FormulaAnd(p[3])
            elif p[2] == 'or':
                p[0] = FormulaOr(p[3])
            elif p[2] == 'not':
                p[0] = FormulaNot(p[3])
        elif len(p) == 6:
            if p[3] == 'imply':
                p[0] = FormulaImply(p[3], p[4])
            elif p[1] == 'exists':
                p[0] = FormulaExists(p[3], p[5])
            elif p[1] == 'forall':
                p[0] = FormulaForall(p[3], p[5])
        elif len(p) == 8:
            if p[2] == 'exists':
                p[0] = FormulaExists(p[4], p[6])
            elif p[2] == 'forall':
                p[0] = FormulaForall(p[4], p[6])
```

```
...
    def p_effects_def(self, p):
        '''effects_def : EFFECT_KEY LPAREN one_eff_formula RPAREN'''
        if p[3] == 'and':
            p[0] = '(and)'
        else:
            p[0] = p[3]

    def p_one_eff_formula(self, p):
        '''one_eff_formula : literal
                           | AND_KEY one_eff_formula_lst
                           | AND_KEY
                           | ONEOF_KEY atomic_eff_lst
                           | WHEN_KEY formula atomic_eff
                           | LPAREN ONEOF_KEY atomic_eff_lst RPAREN
                           | LPAREN WHEN_KEY formula atomic_eff RPAREN
                           | LPAREN FORALL_KEY LPAREN typed_variables_lst
                           RPAREN atomic_eff RPAREN
                           | LPAREN FORALL_KEY LPAREN typed_variables_lst
                           RPAREN LPAREN WHEN_KEY formula atomic_eff
                           RPAREN RPAREN'''
        if len(p) == 2:
            p[0] = p[1]
        elif len(p) == 3:
            if p[1] == 'and':
                p[0] = FormulaAnd(p[2])
            else: p[0] = FormulaOneOf(p[2])
        elif len(p) == 4:
            if p[1] == 'when':
                p[0] = FormulaWhen(p[2], p[3])
        elif len(p) == 5:
            p[0] = FormulaOneOf(p[3])
        elif len(p) == 6:
            p[0] = FormulaWhen(p[3], p[4])
        elif len(p) == 8:
            p[0] = FormulaForall(p[4], p[6])
        elif len(p) == 12:
            nested = FormulaWhen(p[8], p[9])
            p[0] = FormulaForall(p[4], nested)
...
    def p_atomic_eff(self, p):
        '''atomic_eff : literal
                      | AND_KEY literal_lst
```

```
                        | LPAREN AND_KEY literal_lst RPAREN
                        | LPAREN WHEN_KEY formula atomic_eff RPAREN'''
        if len(p) == 2:
            p[0] = p[1]
        elif len(p) == 3:
            if p[1] == 'and':
                p[0] = FormulaAnd(p[2])
        elif len(p) == 5:
            if p[2] == 'and':
                p[0] = FormulaAnd(p[3])
        elif len(p) == 6:
            p[0] = FormulaWhen(p[3], p[4])
...
    def p_literal(self, p):
        '''literal : LPAREN NOT_KEY predicate RPAREN
                      | predicate'''
        if len(p) == 2:
            p[0] = Literal.positive(p[1])
        elif len(p) == 5:
            p[0] = Literal.negative(p[3])

    def p_predicate(self, p):
        '''predicate : LPAREN NAME variables_lst RPAREN
                        | LPAREN EQUALS VARIABLE VARIABLE RPAREN
                        | LPAREN NAME RPAREN
                        | NAME variables_lst
                        | EQUALS VARIABLE VARIABLE
                        | NAME'''
        if len(p) == 2:
            p[0] = Predicate(p[1])
        elif len(p) == 3:
            p[0] = Predicate(p[1], p[2])
        elif len(p) == 4:
            if p[1] == '(':
                p[0] = Predicate(p[2])
            else:
                p[0] = Predicate('=', [p[2], p[3]])
        elif len(p) == 5:
            p[0] = Predicate(p[2], p[3])
        elif len(p) == 6:
            p[0] = Predicate('=', [p[3], p[4]])

    def p_typed_variables_lst(self, p):
```

```
        '''typed_variables_lst : variables_lst HYPHEN type
        typed_variables_lst
                                | variables_lst HYPHEN type'''
        if len(p) == 4:
            p[0] = [Term.variable(name, p[3]) for name in p[1]]
        else:
            p[0] = [Term.variable(name, p[3]) for name in p[1]] + p[4]

    def p_variables_lst(self, p):
        '''variables_lst : VARIABLE variables_lst
                        | VARIABLE'''
        if len(p) == 2:
            p[0] = [p[1]]
        elif len(p) == 3:
            p[0] = [p[1]] + p[2]
...
```

In Listing 4.2, there are all methods definition that are in charge of constructing the syntax tree structure from tokens found by the lexer in the input string document. Then, we have implemented a data structure for all components belonging to the PDDL standard, starting from terms and predicates to the whole domain and problem. In this way, we provide a code more manageable and readable.

At this point, we are going to describe one after the other the implemented data structure of the main PDDL components.

Firstly, variables and constants are represented by the `Term` class in Listing 4.3.

Listing 4.3. The `Term` class.

```
class Term:

    def __init__(self, **kwargs):
        self._name = kwargs.get('name', None)
        self._type = kwargs.get('type', None)
        self._value = kwargs.get('value', None)

    @property
    def name(self):
        return self._name

    @property
    def type(self):
        return self._type

    @property
    def value(self):
```

```
        return self._value
...
    @classmethod
    def variable(cls, name, type=None):
        return Term(name=name, type=type)

    @classmethod
    def constant(cls, value, type=None):
        return Term(value=value, type=type)

    def __str__(self):
        if self.is_variable() and self.is_typed():
            return '{0} - {1}'.format(self._name, self._type)
        if self.is_variable():
            return '{0}'.format(self._name)
        if self.is_constant() and self.is_typed():
            return '{0} - {1}'.format(self._value, self._type)
        if self.is_constant():
            return '{0}'.format(self._value)
```

A `Term` object is instantiated with three optional arguments: a name, a type and a value, respectively. As said before, a term can represent a variable or a constant. In each of these cases, it handles a possible typing and/or a possible value.

Secondly, we have the `Predicate` and the `Literal` classes. The former is used to represent PDDL fluents, whereas the latter distinguishes between positive and negative fluents. Additionally, the `Predicate` class can also handle the equality predicates. In the following Listing we propose a glimpse of both the `Predicate` and the `Literal` classes.

Listing 4.4. The `Predicate` and the `Literal` classes.

```
class Predicate:

    def __init__(self, name, args=[]):
        self._name = name
        self._args = args

    @property
    def name(self):
        return self._name

    @property
    def args(self):
        return self._args[:]
...
    def __str__(self):
```

```
        if self.name == '=':
            return '(=␣{0}␣{1})'.format(str(self._args[0]),
            str(self._args[1]))
        elif self.arity == 0:
            return '('+self.name+')'
        else:
            return '({0}␣{1})'.format(self.name, '␣'.join(map(str,
            self._args)))
...

class Literal:

    def __init__(self, predicate, positiveness):
        self.predicate = predicate
        self.positiveness = positiveness
...
    @classmethod
    def positive(cls, predicate):
        return Literal(predicate, True)

    @classmethod
    def negative(cls, predicate):
        return Literal(predicate, False)

    def get_vars(self):
        return self.predicate.args

    def __str__(self):
        if self.is_positive():
            return str(self.predicate)
        if not self.is_positive() and self.predicate.name == '=':
            lhs = str(self.predicate.args[0])
            rhs = str(self.predicate.args[1])
            return '(not␣(=␣{0}␣{1}))'.format(lhs, rhs)
        if not self.is_positive():
            return '(not␣{})'.format(str(self.predicate))
```

After seeing the data structure for low level objects that we have just presented, we are going to illustrate the implementation of data structure for the rest, namely the `Action` class, the `Domain` class and, finally, the `Problem` class. Here, we skip the implementation of each formula since it simply consists of initializing and representing the corresponding object.

Firstly, we show main methods of the `Action` class in Listing 4.5.

Listing 4.5. The `Action` class.

```python
from fond4ltlfpltl.PDDLparser.formula import FormulaAnd, FormulaOneOf
from fond4ltlfpltl.PDDLparser.literal import Literal
from fond4ltlfpltl.PDDLparser.predicate import Predicate

class Action:

    def __init__(self, name, parameters, preconditions, effects):
        self.name = name #string
        self.parameters = parameters #list
        self.preconditions = preconditions #formula.FormulaXXX
        self.effects = effects #formula.FormulaXXX

    def __str__(self):
        operator_str = '{0}\n'.format(self.name)
        operator_str += '\t:parameters␣({0})\n'.format('␣'.join(map(str,
        self.parameters)))
        operator_str += '\t:precondition␣{0}\n'.format(self.preconditions)
        operator_str += '\t:effect␣{0}\n'.format(self.effects)
        return operator_str

    def add_to_precond(self):
        if isinstance(self.preconditions, FormulaAnd):
            self.preconditions.complete_domain_turn(True)
        else:
            old_formula = self.preconditions
            precond_to_be_added = Literal.positive(Predicate('turnDomain'))
            self.preconditions = FormulaAnd([old_formula,precond_to_be_added])

    def add_to_effect(self):
        if isinstance(self.effects, FormulaAnd):
            self.effects.complete_domain_turn(False)
        else:
            old_formula = self.effects
            effect_to_be_added = Literal.negative(Predicate('turnDomain'))
            self.effects = FormulaAnd([old_formula,effect_to_be_added])

    def add_turn_domain(self):
        self.add_to_precond()
        self.add_to_effect()
```

As we can see from the Listing, the `Action` class contains all main properties, i.e. the name, parameters, preconditions and effects. Moreover, as done for all classes, the `Action` class has its string representation that convert the operator itself in PDDL notation. As last remark, notice the `add_turn_domain` method which performs the addition to each operator (except from `trans` operators) of the domain of the new predicate `turnDomain`. In particular, we remind the reader that `turnDomain` is put positive in preconditions and negative on effects.

Secondly, we describe the two main data structures representing PDDL elements: the `Domain` class and the `Problem` class, respectively. Both represent a complete planning task. In Listing 4.6, we can see the definition of the `Domain` class.

Listing 4.6. The `Domain` class.

```
from fond4ltlfpltl.PDDLparser.formula import *
from fond4ltlfpltl.PDDLparser.action import Action
import copy

class Domain:

    def __init__(self, name, requirements, types, constants, predicates,
    operators):
        self.name = name #string
        self.requirements = requirements #list
        self.types = types #list
        self.constants = constants #list
        self.predicates = predicates #list
        self.operators = operators #list
...
    def add_operators_trans(self, transition_operators):
        for operator in transition_operators:
            self.operators.append(operator)

    def add_predicates(self, parameters, states):
        self.predicates.append('(turnDomain)')
        for state in states:
            self.predicates.append('(q{0}␣{1})'.format(str(state),
            '␣'.join(map(str, parameters))))
...
    def add_precond_effect(self):
        for op in self.operators:
            op.add_turn_domain()

    def get_new_domain(self, parameters, states, transition_operators):
        self.compile_simple_adl()
        self.add_predicates(parameters, states)
        self.add_precond_effect()
        self.add_operators_trans(transition_operators)
        return self
```

```

    def modify_operator(self, op_copy, condition, formula):
        if isinstance(op_copy.preconditions, FormulaAnd):
            op_copy.preconditions.andList.append(condition)
            new_preconditions = op_copy.preconditions.andList
            new_effects = formula
        else:
            new_preconditions = FormulaAnd([op_copy.preconditions,condition])
            new_effects = formula
        new_op = Action(op_copy.name, op_copy.parameters,
         new_preconditions, new_effects)
        return new_op

    def compile_simple_adl(self):
        i = 0
        for op in self.operators:
            if isinstance(op.effects, FormulaWhen):
                # it remains only one operator, but we need to modify it
                condition_formula = op.effects.condition
                statement_formula = op.effects.formula
                new_op = self.modify_operator(copy.deepcopy(op),
                condition_formula, statement_formula)
                self.operators[i] = new_op
                continue
            elif isinstance(op.effects, FormulaAnd):
                no_of_whens = op.effects.count_whens()
                if no_of_whens == 0:
                    continue
                else:
                    pos = self.operators.index(op)
                    new_op_list = self.split_operator(copy.deepcopy(op),
                    no_of_whens)
                    self.operators[pos:pos+1] = new_op_list
            i +=1

    def split_operator(self, op, number):
        '''
        given a simple adl operator it returns
        a list of operators without adl
        '''
        new_op_list = []
        pair_precond_effect = []
        additionals = []
```

```
79         formula = op.effects
80 
81         for item in formula.andList:
82             if isinstance(item, FormulaWhen):
83                 formula_condition = item.condition
84                 formula_statement = item.formula
85                 pair_precond_effect.append([FormulaAnd([op.preconditions,
86                 formula_condition]), formula_statement])
87             else:
88                 additionals.append(item)
89         k = 1
90         for j in range(len(pair_precond_effect)):
91             pair_precond_effect[j][k] = FormulaAnd([pair_precond_effect[j]
92             [k]]+additionals)
93 
94         for u in range(number):
95             new_op = Action(op.name+'-'+str(u), op.parameters,
96              pair_precond_effect[u][0], pair_precond_effect[u][1])
97             new_op_list.append(new_op)
98 
99         return new_op_list
```

As seen in Section 4.2.1, a PDDL domain consists of a name, the list of requirements, the list of types, the list of constants, the list of predicates and, finally, the list of operators. All items of these list, are instantiations of classes previously seen. Regarding the `Domain` class, it comprises a couple of methods that contribute to the creation of the new domain in which we encode temporal goals. Additionally, at line 49, the class provides a method to compile away simple conditional effects in operators. This functionality simply splits operators containing conditional effects as many times as the number of conditional effects appearing in the action. More specifically, for each conditional effect we put the condition in action's preconditions and leave the effect-formula in the action's effect. Finally, we have the `get_new_domain` method, at line 30, that gathers and invokes all methods for obtaining the new domain with encoded temporal goals.

Finally, we present the `Problem` class in Listing 4.7.

Listing 4.7. The `Problem` class.

```
from fond4ltlfpltl.PDDLparser.formula import FormulaOr

class Problem(object):

    new_goal = set()

    def __init__(self, name, domain, objects, init, goal):
        self._name = name
```

```
        self._domain = domain
        self._objects = {}
        for obj in objects:
            self._objects[obj.type] = self._objects.get(obj.type, [])
            self._objects[obj.type].append(str(obj.value))
        self._init = set(map(str, init))
        self._goal = set(map(str, goal))

    @property
    def name(self):
        return self._name

    @property
    def domain(self):
        return self._domain

    @property
    def objects(self):
        return self._objects.copy()

    @property
    def init(self):
        return self._init.copy()

    @property
    def goal(self):
        return self._goal.copy()

    def __str__(self):
        problem_str = '(define (problem {0})\n'.format(self._name)
        problem_str += '\t(:domain {0})\n'.format(self._domain)
        problem_str += '\t(:objects'
        for type, objects in self._objects.items():
            problem_str += ' {0} - {1}'.format(' '.join(sorted(objects)),
            type)
        problem_str += ')\n'
        problem_str += '\t(:init {0})\n'.format(' '.join(sorted(
        self._init)))
        problem_str += '(:goal (and {0}))\n'.format(
        ' '.join(sorted(self.new_goal)))
        problem_str += ')'
        return problem_str
```

```
    def make_new_init(self, obj_list):
        self._init.add('(turnDomain)')
        if obj_list:
            self._init.add('(q1␣{0})'.format('␣'.join(obj_list)))
        else:
            self._init.add('(q1)')
        return self._init

    def make_new_goal(self, final_states, obj_list):
        self.new_goal.add('(turnDomain)')
        if len(final_states) > 1:
            or_list = []
            for state in final_states:
                if obj_list:
                    or_list.append('(q{0}␣{1})'.format(str(state),
                    '␣'.join(obj_list)))
                else:
                    or_list.append('(q{0})'.format(str(state)))
            new_formula = FormulaOr(or_list)
            self.new_goal.add(str(new_formula))
        else:
            if obj_list:
                self.new_goal.add('(q{0}␣{1})'.format(final_states[0],
                '␣'.join(obj_list)))
            else:
                self.new_goal.add('(q{0})'.format(final_states[0]))

    def get_new_problem(self, final_states, symbols_list):
        obj_list = self.extract_object_list(symbols_list)
        self.objects_are_upper(obj_list)
        self.make_new_init(obj_list)
        self.make_new_goal(final_states, obj_list)
        return self

    def extract_object_list(self, symbols_list):
        already_seen = set()
        obj_list = []
        for symbol in symbols_list:
            if symbol.objects:
                for obj in symbol.objects:
                    if obj not in already_seen:
                        already_seen.add(obj)
                        obj_list.append(obj)
```

```
95                     else:
96                         pass
97             else:
98                 continue
99         return obj_list
100
101     def objects_are_upper(self, objects):
102         for value_list in self.objects.values():
103             for val in value_list:
104                 if val.isupper() and val.lower() in objects:
105                     objects[objects.index(val.lower())] =
106                     objects[objects.index(val.lower())].upper()
107                 else:
108                     pass
```

The main methods are:

- `make_new_init` (line 52);
- `make_new_goal` (line 60);
- `get_new_problem` (line 79).

Having the informations about the DFA encoding, the first method computes the new initial specification by adding the `turnDomain` predicates and the initial state predicate of the automaton. Likewise, the second method assembles the new goal specification accordingly with the theory illustrated in Section 4.3.2. Finally, the third method simply returns the new problem encoded in standard PDDL.

4.4.3 Automa

In this section, we illustrate the code that handles extended temporal goals treated as automata. More specifically, we talk about the generation of DFA corresponding to the LTL$_f$/PLTL goal, the parsing and managing of such generated DFA showing all implemented data structures.

First of all, as stated before, given an LTL$_f$/PLTL goal formula, we employ the LTL$_f$2DFA tool to generate the DFA. Secondly, as soon as the DFA is generated, we parse it with the `aparser.py` module and then, we instantiate an `Automa` object for it. In the following, we will discuss more about the `Automa` class, since the `aparser.py` module is pretty much the same discussed in Section 5.4.3.

Listing 4.8. The `Automa` class.

```
...
class Automa:
    def __init__(self, alphabet, states, initial_state, accepting_states,
     transitions):
```

```
        self.alphabet = alphabet
        self.states = states
        self.initial_state = initial_state
        self.accepting_states = accepting_states
        self.transitions = transitions
        self.trans_by_dest = self.group_conditions_by_consequence()
        self.validate()
...
    def validate(self):
        self.validate_initial_state()
        self.validate_accepting_states()
        self.valide_transition_start_states()
        self.validate_input_symbols()
        return True

    def __str__(self):
        automa = 'alphabet:␣{}\n'.format(str(self.alphabet))
        automa += 'states:␣{}\n'.format(str(self.states))
        automa += 'init_state:␣{}\n'.format(str(self.initial_state))
        automa += 'accepting_states:␣{}\n'.format(str(self.accepting_states))
        automa += 'transitions:␣{}'.format(str(self.transitions))
        return automa

    def create_operators_trans(self, domain_predicates, grounded_symbols):
        new_operators = []
        my_predicates = []
        for symbol in grounded_symbols:
            my_predicates.append(symbol.name)
        (parameters, obj_mapping) =  self.compute_parameters(domain_predicates,
        grounded_symbols)
        vars_mapping = self.compute_varsMapping(grounded_symbols,
         obj_mapping)
        my_variables = self.compute_variables(parameters)
        counter = 0
        for destination, source_action in self.trans_by_dest.items():
            if source_action:
                fluents_list_precond = self.compute_preconditions(source_action,
                 vars_mapping, my_predicates, my_variables)
                if isinstance(fluents_list_precond, FormulaAnd):
                    new_precondition = fluents_list_precond
                else:
                    new_precondition = FormulaAnd([fluents_list_precond]
                    + [Literal.negative(Predicate('turnDomain'))])
```

```
                new_effects = self.compute_effects(destination, my_variables)
                new_operators.append(Action('trans-'+str(counter),
                 parameters, new_precondition, new_effects))
            else:
                pass
            counter += 1
        return (new_operators, parameters)

    def compute_type(self, all_predicates, name, position):
        for predicate in all_predicates:
            if predicate.name == name:
                if predicate.args:
                    return (predicate.args[position].name +
                     ''.join(random.choices(string.digits, k=2)),
                      predicate.args[position].type)
                else:
                    raise ValueError('[ERROR]: Please check the formula')
            else:
                pass

    def compute_parameters(self,domain_predicates, grounded_symbols):
        objs_set = set()
        obj_mapping = {}
        parameters = []
        for symbol in grounded_symbols:
            if symbol.objects:
                i = 0
                for obj in symbol.objects:
                    if obj not in objs_set:
                        objs_set.add(obj)
                        (name_var, type_) = self.compute_type(domain_predicates,
                        symbol.name, i)
                        obj_mapping[obj] = [name_var, type_]
                        parameters.append(Term.variable(name_var, type_))
                    else:
                        pass
                    i += 1
        return (parameters, obj_mapping)

    def compute_varsMapping(self, grounded_symbols, obj_mapping):
        temp = []
        vars_mapping = {}
        for symbol in grounded_symbols:
```

```
            if symbol.objects:
                for obj in symbol.objects:
                    temp.append((obj_mapping[obj][0], obj_mapping[obj][1]))
            else:
                vars_mapping[symbol] = []
            vars_mapping[symbol] = temp
            temp = []
        return vars_mapping

    def compute_preconditions(self, source_action, vars_mapping,
     predicates_name, variables):
        if len(source_action) == 1:
            if self.get_automaton_formula(vars_mapping, predicates_name,
             source_action[0][1]) == []:
                formula = Literal.positive(Predicate('q'+
                str(source_action[0][0]), variables))
            else:
                automaton_state = [Literal.positive(Predicate('q'+
                str(source_action[0][0]), variables))]
                formula = FormulaAnd(automaton_state+
                self.get_automaton_formula(vars_mapping, predicates_name,
                 source_action[0][1])+[Literal.negative(Predicate('turnDomain'))])
        else:
            formula = FormulaOr(self.get_or_conditions(vars_mapping,
             predicates_name, variables, source_action))
        return formula

    def compute_effects(self, destination, variables):
        negated_states = []
        for state in self.states:
            if state != destination:
                negated_states.append(Literal.negative(Predicate('q'+
                str(state), variables)))
            else:
                pass
        automaton_destination = [Literal.positive(Predicate('q'+
        str(destination), variables))]
        turnDomain = [Literal.positive(Predicate('turnDomain'))]
        formula = FormulaAnd(automaton_destination+negated_states+turnDomain)
        return formula

    def get_or_conditions(self, vars_mapping, predicates_name, variables,
     source_action_list):
```

```
        items = []
        for source, action in source_action_list:
            formula = self.get_automaton_formula(vars_mapping,
             predicates_name, action)
            if formula == []:
                items.append(Literal.positive(Predicate('q'+str(source),
                 variables)))
            else:
                automaton_state = [Literal.positive(Predicate('q'+
                str(source), variables))]
                items.append(FormulaAnd(automaton_state+
                self.get_automaton_formula(vars_mapping, predicates_name,
                action)))
        return items

    def get_automaton_formula(self, vars_mapping, predicates_name, action):
        temp = []
        i = 0
        for char in action:
            if char == '1':
list(vars_mapping)]:
                temp.append(Literal.positive(Predicate(predicates_name[i],
                 [x[0] for x in vars_mapping[list(vars_mapping)[i]]] )))
temp.append(Literal.positive(Predicate(predicates_name[i])))
            elif char == '0':
                temp.append(Literal.negative(Predicate(predicates_name[i],
                 [x[0] for x in vars_mapping[list(vars_mapping)[i]]] )))
temp.append(Literal.negative(Predicate(predicates_name[i])))
            else:
                pass
            i += 1
        return temp

    def compute_variables(self, parameters_list):
        my_variables = []
        for param in parameters_list:
            my_variables.append(param.name)
        return my_variables

    def create_dict_by_destination(self):
        trans_by_dest = {}
        for state in self.states:
            trans_by_dest[state] = []
```

```
177         return trans_by_dest
178 
179     def group_conditions_by_consequence(self):
180         group_by_dest = self.create_dict_by_destination()
181         for source, trans in self.transitions.items():
182             i = 0
183             for dest in trans.values():
184                 group_by_dest[dest].append((source, list(trans.keys())[i]))
185                 i +=1
186         return group_by_dest
```

In Listing 4.8, we have the main data structure representing the DFA generated by LTL$_f$2DFA. As soon as an object is instantiated, transitions are grouped by destination state by a method at line 10. This operation is useful to get a compact representation of each `trans` operator created. We recall that, the `trans` operator, created in Section 4.3.2, is then split in multiple `trans` operators because not all planners support conditional effects.

The main method of the `Automa` class is obviously the `create_operators_trans` which gather all other methods. It takes as input the domain predicates and the grounded symbols of the goal formula. Moreover, the method consists of the operations explained theoretically. In particular, we have:

- it computes all parameters involved by the grounded symbols of the formula, at line 33, using the `compute_parameters` method;
- it implements the function f, defined in 4.11, mapping objects to variables at line 35;
- it computes variables;
- at line 39, it builds preconditions and effects for each transition of the DFA;
- for each `trans` operator created, it is added to a the list of trans operators.

In the next Section, we will see the main module, which implements all the high level procedure illustrated previously in Section 4.3.

4.4.4 Main Module

Here, we describe the main module of the FOND4LTL$_f$/PLTL package. It is called `fond4ltlfpltl.py` and it contains the principal logic covering the FOND4LTL$_f$/PLTL theoretical approach illustrated in Section 4.3. In Listing 4.9, we show the implementation.

Listing 4.9. The `fond4ltlfpltl.py` module.

```
from fond4ltlfpltl.PDDLparser.parser import MyParser
from ltlf2dfa.Translator import Translator
```

```
from ltlf2dfa.DotHandler import DotHandler
from fond4ltlfpltl.AutomaParser.symbol import Symbol
from fond4ltlfpltl.AutomaParser.aparser import parse_dot
import argparse, os, subprocess, copy, re

args_parser = argparse.ArgumentParser(...)

params = vars(args_parser.parse_args())
pddl_parser = MyParser()

try:
    with open(params['<planning_domain>'], 'r') as f:
        domain = f.read()
        f.close()
    parsed_domain = pddl_parser(domain)
except:
    raise ValueError('[ERROR]:␣Could␣not␣parse␣domain')

try:
    with open(params['<planning_problem>'], 'r') as f:
        problem = f.read()
        f.close()
    parsed_problem = pddl_parser(problem)
except:
    raise ValueError('[ERROR]:␣Could␣not␣parse␣problem')
...
try:
    translator = Translator(params['<goal_formula>'])
    translator.formula_parser()
    translator.translate()
    translator.createMonafile(False)
    translator.invoke_mona()
    dot_handler = DotHandler()
    dot_handler.modify_dot()
    dot_handler.output_dot()
    dfa_automaton = parse_dot("automa.dot")
    operators_trans, parameters = dfa_automaton.create_operators_trans(...)
    os.remove('automa.mona')
    os.remove('automa.dot')
except:
    os.remove('automa.mona')
    os.remove('automa.dot')
    raise ValueError('[ERROR]:␣Could␣not␣create␣DFA')
```

```
46
47 old_domain = copy.deepcopy(parsed_domain)
48
49 new_domain = parsed_domain.get_new_domain(...)
50 new_problem = parsed_problem.get_new_problem(...)
51
52 try:
53     with open("./new-dom.pddl", 'w+') as dom:
54         dom.write(str(new_domain))
55         dom.close()
56     with open("./new-prob.pddl", 'w+') as prob:
57         prob.write(str(new_problem))
58         prob.close()
59 except:
60     raise IOError('[ERROR]:␣Something␣wrong␣occurred')
61
62 ...
```

To begin with, the main module accepts as input a PDDL domain, a PDDL problem and a goal formula. All these arguments are handled by the argument parser at line 8.

After gathered all inputs, we proceed by parsing both the PDDL domain and the PDDL problem (lines 17 and 25). The parsing phase is carried out by module illustrated in Section 4.4.2.

Then, we translate the LTL$_f$/PLTL goal formula into the corresponding DFA employing LTL$_f$2DFA. The code that is in charge of doing that starts at line 30. Once the DFA is built, we can construct the new `trans` operator at line 39.

Consequently, we put all things together and we create the new domain $\mathcal{D}'$ and the problem $\mathcal{P}'$ (lines 49 and 50). Finally, we invoke the FOND-SAT planner feeding it with $\mathcal{D}'$ and $\mathcal{P}'$. If a strong plan is found we obtain it in a file on the working directory.

4.5 Results

In this Section, we show an execution of the FOND4LTL$_f$/PLTL tool as example of result. In particular, we show an execution involving a PLTL goal. We go step-by-step through the solution and we pay attention to the `trans` operator.

We remind the reader that FOND$_{sp}$ planning for PLTL goals is interpreted as reaching a final state such that the history leading to such a state satisfies the given PLTL formula. For instance, here we show an execution of the FOND4LTL$_f$/PLTL tool on the Triangle Tireworld planning task partly illustrated in Example 4.7. Indeed, the original domain is the same of the one in Example 4.7 whereas the original initial state is as follows:

```
(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
```

```
  (:init (vehicle-at l11)
    (road l11 l12) (road l12 l13) (road l11 l21) (road l12 l22)
    (road l21 l12) (road l22 l13) (road l21 l31) (road l31 l22)
    (spare-in l21) (spare-in l22) (spare-in l31)
    (not-flattire))
  (:goal (vehicle-at l13))
)
```

In this case, we choose the PLTL goal formula $\varphi = \mathit{vehicleAt}(l13) \wedge \diamondminus(\mathit{vehicleAt}(l23))$. Such a formula means *reach location l13 passing through location l23 at least once.* The DFA corresponding to φ, generated with LTL$_f$2DFA, is depicted in Figure 4.6.

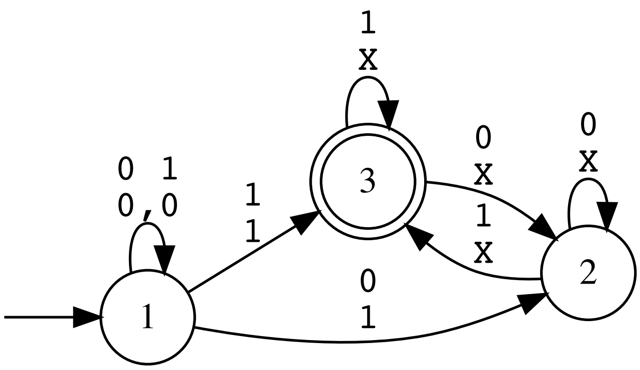


Figure 4.6. The DFA corresponding to φ

After the execution of FOND4LTL$_f$/PLTL, we obtain the following:

1. a new planning domain;
2. a new planning problem;
3. a transition system showing all policies, if found.

Firstly, the new domain $\mathcal{D}'$ is:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicleat ?loc - location) (spare-in ?loc - location)
  (road ?from - location ?to - location) (not-flattire) (turnDomain)
  (q2 ?loc30 - location ?loc53 - location)
  (q1 ?loc30 - location ?loc53 - location)
  (q3 ?loc30 - location ?loc53 - location))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicleat ?from) (road ?from ?to)
    (not-flattire) (turnDomain))
```

```
    :effect (and (oneof (and (vehicleat ?to)
    (not (vehicleat ?from))) (and (vehicleat ?to)
    (not (vehicleat ?from)) (not (not-flattire))))
    (not (turnDomain)))
  )
  (:action changetire
    :parameters (?loc - location)
    :precondition (and (spare-in ?loc) (vehicleat ?loc)
    (turnDomain))
    :effect (and (not (spare-in ?loc)) (not-flattire)
    (not (turnDomain)))
  )
  (:action trans-0
    :parameters (?loc30 - location ?loc53 - location)
    :precondition (and (or (and (q1 ?loc30 ?loc53)
    (not (vehicleat ?loc30)) (vehicleat ?loc53))
    (and (q2 ?loc30 ?loc53) (not (vehicleat ?loc30)))
    (and (q3 ?loc30 ?loc53) (not (vehicleat ?loc30))))
    (not (turnDomain)))
    :effect (and (q2 ?loc30 ?loc53)
    (not (q1 ?loc30 ?loc53)) (not (q3 ?loc30 ?loc53))
    (turnDomain))
  )
  (:action trans-1
    :parameters (?loc30 - location ?loc53 - location)
    :precondition (and (or (and (q1 ?loc30 ?loc53)
    (not (vehicleat ?loc30)) (not (vehicleat ?loc53)))
    (and (q1 ?loc30 ?loc53) (vehicleat ?loc30)
    (not (vehicleat ?loc53)))) (not (turnDomain)))
    :effect (and (q1 ?loc30 ?loc53)
    (not (q2 ?loc30 ?loc53)) (not (q3 ?loc30 ?loc53))
    (turnDomain))
  )
  (:action trans-2
    :parameters (?loc30 - location ?loc53 - location)
    :precondition (and (or (and (q1 ?loc30 ?loc53)
    (vehicleat ?loc30) (vehicleat ?loc53))
    (and (q2 ?loc30 ?loc53) (vehicleat ?loc30))
    (and (q3 ?loc30 ?loc53) (vehicleat ?loc30)))
    (not (turnDomain)))
    :effect (and (q3 ?loc30 ?loc53)
    (not (q2 ?loc30 ?loc53)) (not (q1 ?loc30 ?loc53))
    (turnDomain))
```

```
  )
)
```

Secondly, the new problem $\mathcal{P}'$ is:

```
(define (problem triangle-tire-1)
        (:domain triangle-tire)
        (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
        (:init (not-flattire) (q1 l13 l23) (road l11 l12) (road l11 l21)
        (road l12 l13) (road l12 l22) (road l21 l12) (road l21 l31)
        (road l22 l13) (road l22 l23) (road l23 l13) (road l31 l22)
        (spare-in l21) (spare-in l22) (spare-in l23) (spare-in l31)
        (turnDomain) (vehicleat l11))
(:goal (and (q3 l13 l23) (turnDomain)))
)
```

Finally, feeding FOND-SAT with $\mathcal{D}'$ and $\mathcal{P}'$ we obtain the following transition system depicted in Figure 4.7, thanks to another Python script, developed in this thesis, for converting a written policy into a graph.

A plan is whatever path from $n0$ leading to state ng. Moreover, as we can see from Figure 4.7, there is a perfect alternation between domain's actions and the `trans` action. Additionally, the above-mentioned script could also remove transitions involving the `trans` action getting the final plan. We can see it in Figure 4.8.

4.6 Summary

In this chapter, we have faced the problem of FOND Planning for LTL$_f$/PLTL goals. In particular, we have proposed a new solution, called FOND4LTL$_f$/PLTL, that essentially reduces the problem to a "classical" FOND planning problem. This has been possible thanks to our LTL$_f$2DFA Python tool which has been employed for the encoding of temporally extended goals into standard PDDL. Then, we have also described in details the FOND4LTL$_f$/PLTL implementation, highlighting all its main features. Finally, we have seen examples of execution results.

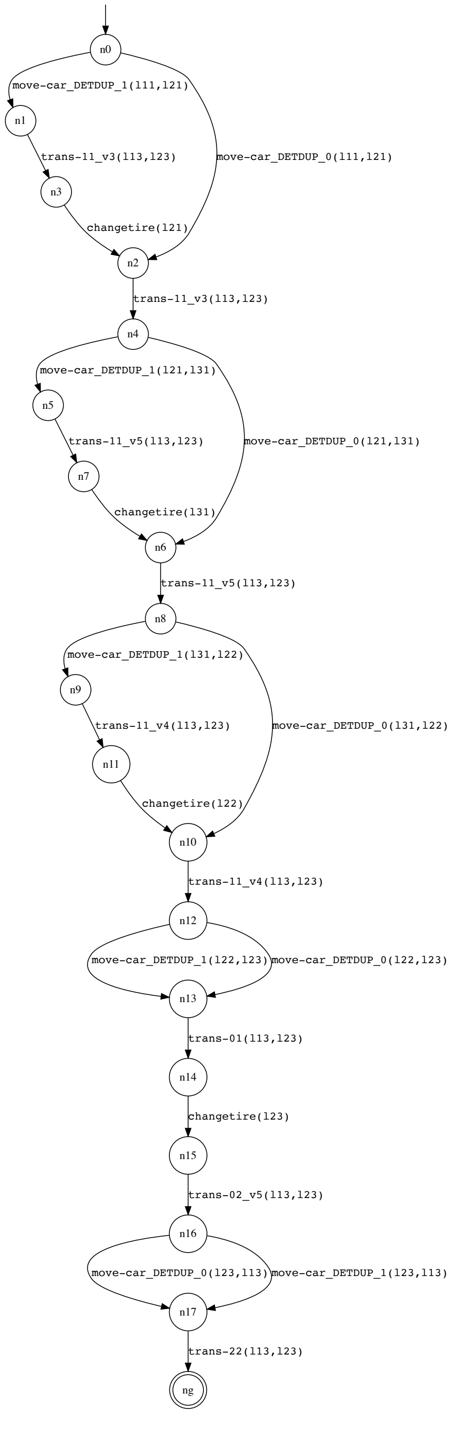


Figure 4.7. The transition system showing policies found with the `trans` operator

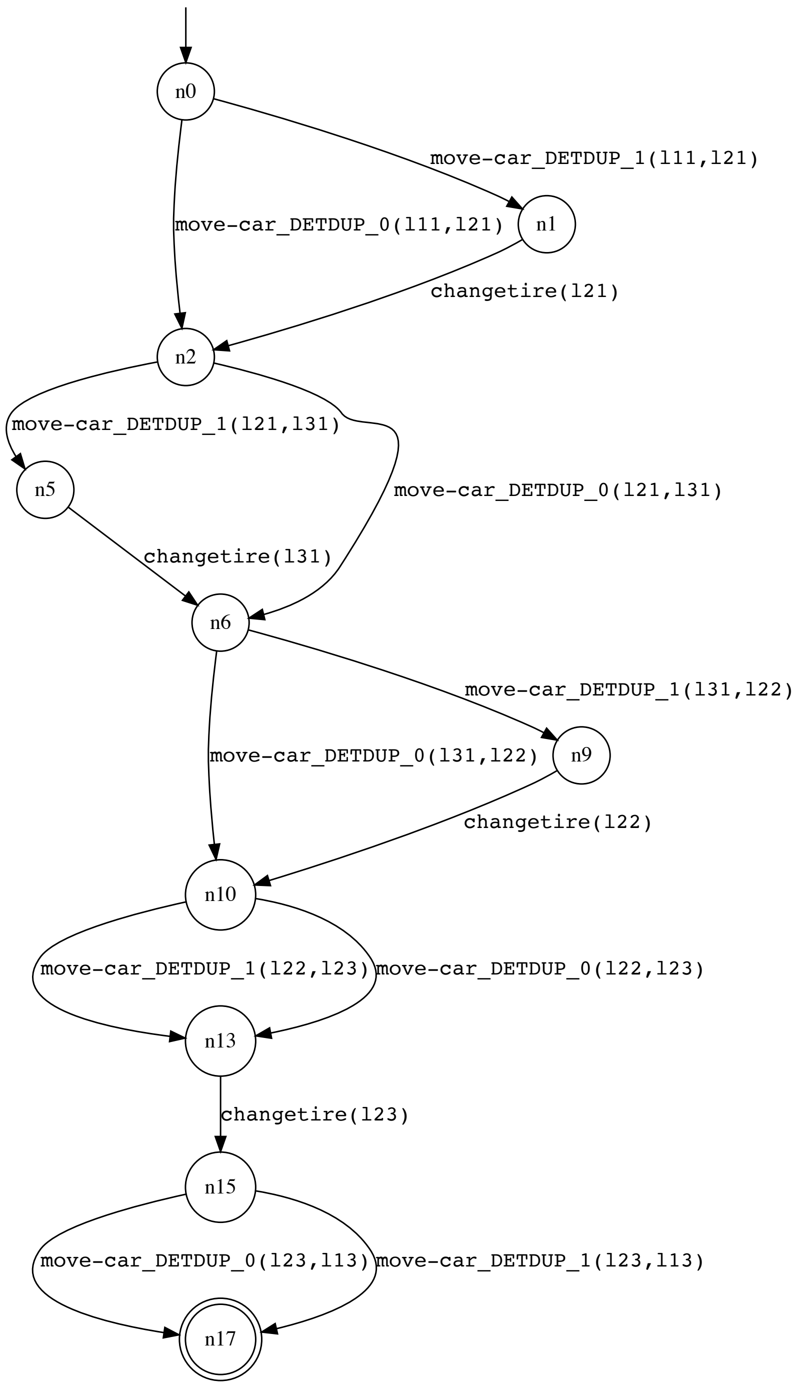


Figure 4.8. The transition system showing policies found without the `trans` operator

Chapter 5

Janus

In this chapter, we will illustrate how our tool LTL_f2DFA presented in Chapter 3 can be efficiently employed in the field of Business Process Management, with particular attention to Declarative Process Mining. First of all, we will formally describe the theoretical framework of declarative process mining. We will give a generalization of the Janus algorithm in (Cecconi et al., 2018), introducing a new theorem that enlarges concepts of reactive constraints and separated formulas and we will illustrate the Janus algorithm in (Cecconi et al., 2018) modified accordingly with the new proposed theory. Then, in this context, we will thoroughly describe the implementation of this new version of the Janus algorithm, employing our tool LTL_f2DFA, for computing the interestingness degree of traces in real event logs. Finally, we will provide such a computation for a real log as an example.

5.1 Declarative Process Mining

In this section, we will present the theoretical framework of Business Process Management focusing our attention to declarative process mining. We will extend what described in Chapter 2 providing all additional concepts, definitions and theorems necessary to clearly understand the context.

Business Process Management (BPM) deals with discovering, modeling, analyzing and managing business processes in order to measure their productivity and to improve their performance. These tasks are carried out thanks to logging facilities that, nowadays, all BPM systems have. The extraction and the validation of temporal constraints from event logs (i.e. multi-sets of finite traces) are techniques consisting declarative process mining (Montali, 2010). Temporal constraints are expressed using LTL_f and/or PLTL and refers to activities present in traces. In the following, we will formally introduce what event logs and DECLARE (Pesic, 2008) are. Another important aspect to notice is that these constraints are meant to be checked upon the activation satisfying specific conditions. For these reasons, they are referred as *reactive constraints*.

5.1.1 Event Logs

The event log is a collection of meaningful data that is the entry point for the consequent process mining. Formally, we consider this meaningful data expressed as a multiple traces containing a sequence of events belonging to the alphabet of symbols Σ. A single trace can be represented as $t = \langle e_1, e_2, \ldots, e_n \rangle$ where e_i is the event occurring at instant i and $n \in \mathbb{N}$ is the length of the trace t. Now, we can give the following definition:

Definition 5.1. An event log $\mathcal{L}$ is defined as $\mathcal{L} = \{t_1, \ldots, t_m\} \in \mathbb{M}(\Sigma^*)$ is a multi-set of traces t_j with $1 \leq j \leq m$, where $m \in \mathbb{N}$.

To better indicate the *multiplicity* of traces in $\mathcal{L}$, we can denote it as a superscript compacting the notation. For example, t_2^{10} stands for trace t_2 occurs 10 times in $\mathcal{L}$.

Example 5.1. $\mathcal{L} = \{t_1^{25}, t_2^{10}, t_3^{15}, t_4^{20}, t_5^{5}, t_6^{10}\}$ is an event log of 85 traces, defined over the alphabet $\Sigma = \{a, b, c, \ldots, i\}$. In $\mathcal{L}$ we have the following traces:

$$
\begin{aligned}
t_1 &= \langle d, f, a, f, c, a, f, b, a, f \rangle \\
t_2 &= \langle f, e, d, c, b, a, g, h, i \rangle \\
t_3 &= \langle a, d, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, c \rangle \\
t_4 &= \langle d, b, a, b \rangle \\
t_5 &= \langle a, d, a, c, a \rangle \\
t_6 &= \langle b, c, d, e \rangle
\end{aligned}
$$

Furthermore, the event e_i occurring at instant i is denoted by $t(i)$, whereas the segment of t (i.e. the sub-trace) ranging from instant i to instant j, where $1 \leq i \leq j \leq n$ is denoted by $t_{[i:j]}$.

Apart from the formal model of event logs, we have real-world event logs that are logs with real data coming from different kind of data sources (e.g. databases, transaction logs, audit log, etc.). All available tools are evaluated against real-world logs or synthetic logs, i.e. automatically generated logs that mimic real logs in shape and content. In practice, as we will see in the Section 5.4, the main way of representing real logs is the eXtensible Event Stream (XES) Standard[1], which is based on the well known XML.

5.1.2 DECLARE

DECLARE is a language concerning declarative process modeling (Pesic, 2008) and consisting of standard templates based on (Dwyer et al., 1999) that was introduced to simplify the complexity of constraints semantics. Indeed, DECLARE constraints are expressed in LTL$_f$, but we will extend LTL$_f$ with Past temporal operators (LTLp$_f$) for capturing also past modalities. In Table 5.1, we can see what are the corresponding LTL$_f$ or LTLp$_f$ formulas for the most important DECLARE constraints.

Parameters in a template define tasks and they occurs as events in traces. In Example 5.2 we provide a glimpse of DECLARE patterns.

[1]http://www.xes-standard.org

Table 5.1. The most important DECLARE constraints expressed as LTLp$_f$ formulas and *reactive constraints.*

DECLARE **constraint**	LTL$_f$ **expression**	**RCon**
PARTICIPATION(a)	$\Diamond a$	$t_{start} \mapsto \Diamond a$
INIT(a)	a	$t_{start} \mapsto a$
END(a)	$\Box\Diamond a$	$t_{end} \mapsto a$
RESPONDEDEXISTENCE(a,b)	$\Diamond a \Rightarrow \Diamond b$	$a \mapsto (\Diamond b \vee \Diamondminus b)$
RESPONSE(a,b)	$\Box(a \Rightarrow \Diamond b)$	$a \mapsto \Diamond b$
ALTERNATERESPONSE(a,b)	$\Box(a \Rightarrow \Diamond b) \wedge \Box(a \Rightarrow \bigcirc(\neg a \bullet b))$	$a \mapsto \bigcirc(\neg a\, \mathcal{U}\, b)$
CHAINRESPONSE(a,b)	$\Box(a \Rightarrow \Diamond b) \wedge \Box(a \Rightarrow \bigcirc b)$	$a \mapsto \bigcirc b$
PRECEDENCE(a,b)	$\neg b \bullet a$	$b \mapsto \Diamondminus a$
ALTERNATEPRECEDENCE(a,b)	$(\neg b \bullet a) \wedge \Box(a \Rightarrow \bigcirc(\neg b \bullet a))$	$b \mapsto \ominus(\neg b\, \mathcal{S}\, a)$
CHAINPRECEDENCE(a,b)	$(\neg b \bullet a) \wedge \Box(\bigcirc b \Rightarrow a)$	$b \mapsto \ominus a$

Example 5.2. Interesting DECLARE templates (Maggi et al., 2013)

- PRECEDENCE(a,b) means *if b occurs then a occurs before b.*
- RESPONCE(a,b) means *if a occurs then eventually b occurs after a.*
- CHAINPRECEDENCE(a,b) means *the occurrence of b imposes a to occur immediately before.*
- ALTERNATERESPONCE(a,b) means *if a occurs then eventually b occurs after a without other occurrences of a in between.*

In addition, one can create his own DECLARE patterns tailored for his purposes. In this way, the DECLARE standard template can be customized.

A given DECLARE constraint is verified over traces and those traces *satisfy* it if they do not *violate* it. Here, it is important to notice that these constraints are prone to the principle of *ex falso quod libet*, namely they can be satisfied even without being activated. This represents a big issue for process mining because mining techniques might misunderstand the actual behavior of a process. The solution to this problem is to compute whether a constraint is satisfied or not only upon activation. However, we will see later how to overcome this problem in the Section 5.2.

Now, we give some definitions:

Definition 5.2. (Gabbay, 1989) Given an LTLp$_f$ formula φ, we call it *pure past* formula ($\varphi^{\blacktriangleleft}$) if it consists of only past operators; *pure present* formula ($\varphi^{\blacktriangledown}$) if it has not any temporal operators; *pure future* formula ($\varphi^{\blacktriangleright}$) if it consists of only future operators.

Example 5.3. Pure formulas:

- $\boxminus(a \Rightarrow \Diamondminus b)$ is a **pure past** formula;
- $a \Rightarrow (b \wedge c)$ is a **pure present** formula

- $\Box(a \Rightarrow \bigcirc b)$ is a **pure future** formula

The separation of an LTLp$_f$ formula to pure past/present/future formulas allows to conduct the analysis on sub-traces (i.e. one referring to the past and the other referring to the future) upon the activation. This is also known as bi-directional on-line analysis. To this extent, we rely on the Separation Theorem stated as follows:

Theorem 5.1. *(Gabbay, 1989) Any propositional temporal formula φ can be rewritten as a boolean combination of pure temporal formulas.*

Therefore, following Theorem 5.1, we can give the Definition of *separated formula* as follows:

Definition 5.3. Let φ an LTLp$_f$ formula over Σ. A temporal separation is a function $\mathcal{S}$: LTLp$_f$ $\rightarrow 2^{\text{LTLp}_f \times \text{LTLp}_f \times \text{LTLp}_f}$ such that: $\mathcal{S}(\varphi) = \{(\varphi^{◀}, \varphi^{▼}, \varphi^{▶})_1, \ldots, (\varphi^{◀}, \varphi^{▼}, \varphi^{▶})_m\}$ such that:

$$\varphi \equiv \bigvee_{j=1}^{m} (\varphi^{◀} \wedge \varphi^{▼} \wedge \varphi^{▶})_j \tag{5.1}$$

where $\varphi^{◀}$, $\varphi^{▼}$ and $\varphi^{▶}$ are pure formulas over Σ as in Definition 5.2.

Notice that Equation 5.1 is a disjunction of conjunction. Moreover, each triple consisting the image function of $\mathcal{S}(\varphi)$ is generally called *separated formula.* In the following, we give an example of separated formula.

Example 5.4. The separated formulas for $(\ominus a \vee \Diamond b)$ is:

$$(\ominus a \wedge True \wedge True) \bigvee (True \wedge True \wedge \Diamond b)$$

Since the Janus algorithm relies on the construction of the automata for separated LTLp$_f$ formulas, we will refer to notions explained previously in Section 2.4. The crucial point is that given a separated LTLp$_f$ formula φ we can build a minimum DFA that *accepts* all and only the traces satisfying formula φ.

In the following sections, first we will describe the Janus approach in (Cecconi et al., 2018) by giving fundamentals definitions and theorems. Secondly, we will illustrate in details our generalized approach highlighting major differences with respect to the original one. Then, we will illustrate the modified algorithm and its practical implementation.

5.2 The Janus Approach

Declarative process modeling defines a list of DECLARE constraints to be satisfied during the execution of the process model. These constraints are of a reactive nature in the sense that the occurrence of some task bounds the occurrence of other activities. As anticipated in the previous Section, this kind of behavior might lead to the principle of *ex falso quod libet*, namely a constraint can be satisfied even though it is never activated. In this Section, we describe the Janus approach (Cecconi et al., 2018) that solves this problem allowing the user to indicate the activation condition for the constraint directly in the constraint. In this way, constraints are activated only if the activation condition holds. Therefore, we can refer to these constraints as *reactive constraints* (RCon).

Definition 5.4. Given an alphabet Σ, let $\alpha \in \Sigma$ be an *activation* and φ be an $\textsc{ltlp}_f$ formula over Σ. A Reactive Constraint (RCon) Ψ is a pair (α, φ), denoted as $\Psi \doteq \alpha \mapsto \varphi$. We represent all the set of RCons over Σ as $\mathcal{R}$.

Hereafter, we will assume traces, automata, $\textsc{ltlp}_f$ formulas and RCons to be defined over the same alphabet Σ. In addition, in Table 5.1, we can see that DECLARE constraints can be converted in RCons. In Definition 5.4, we have seen that α in an RCon is called the *activation*. Indeed, it actually *activates* the corresponding constraint. As in Cecconi et al. (2018), we give the following definitions that are the core concepts upon which the Janus algorithm is built.

Definition 5.5. Given a finite trace $t \in \Sigma$ of length n, and an instant i, with $1 \leq i \leq n$, an RCon $\Psi \doteq \alpha \mapsto \varphi$ is activated at i if $t, i \models \alpha$. Thus, the event $t(i)$ is called the *activator* of Ψ. A trace in which at least an activator of Ψ exists, is *triggering* for Ψ.

Definition 5.6. Given a finite trace $t \in \Sigma$ of length n, an instant i, with $1 \leq i \leq n$, an RCon $\Psi \doteq \alpha \mapsto \varphi$, Ψ is *interesting fulfilled* at i if $t, i \models \alpha$ and $t, i \models \varphi$. The RCon is *violated* at instant i if $t, i \models \alpha$ and $t, i \not\models \varphi$. Otherwise, the RCon is unaffected.

Definition 5.6 is called *interesting fulfilment*, since it formally solves the problem of constraint satisfaction without activation by identifying only those events where the activation condition holds and the RCon is fulfilled. Therefore, every time an event is the activator of an RCon, the RCon is checked for fulfilment. After these two definitions we have to define also an empirical method to compute the *interesting fulfilment* of an RCon for an event log.

Definition 5.7. Given a finite trace $t \in \Sigma$ of length n and an RCon $\Psi \doteq \alpha \mapsto \varphi$, the *interestingness degree* function $\zeta : \mathcal{R} \times \Sigma^* \to [0, 1] \subseteq \mathbb{R}$ is defined as follows:

$$\zeta(\Psi, t) = \begin{cases} \dfrac{|\{i : t, i \models \alpha \text{ and } t, i \models \varphi\}|}{|\{i : t, i \models \alpha\}|}, & \text{if } |\{i : t, i \models \alpha\}| \neq 0; \\ 0, & \text{otherwise} \end{cases}$$

Intuitively, the $\zeta(\Psi, t)$ function measures how many times the RCon Ψ is interesting fulfilled with respect to the total number of activations within the trace t. Now, we give an example to better capture the concepts just defined.

Example 5.5. Let us consider the RCon $\Psi = b \mapsto \diamondminus a$ and traces in the Example 5.1, we have the following:

- Ψ is activated in trace t_1 by $t_1(8)$, in t_2 by $t_2(5)$, in t_4 by $t_4(2)$ and $t_4(4)$ and in t_6 by $t_6(1)$. Hence, t_1, t_2, t_4 and t_6 are *triggering* for Ψ, while Ψ is not activated in t_3 and t_5.

- Ψ is *interestingly fulfilled* by $t_1(8)$ in t_1, by only $t_4(4)$ in t_4. Moreover, Ψ is *violated* by $t_2(5)$ in t_2, by $t_4(2)$ in t_4 and by $t_6(1)$ in t_6. Finally, it is *unaffected* both in t_3 and t_5.
- The *interestingness degree* of Ψ in t_1 is $\zeta(\Psi, t_1) = 1$, since it is activated and fulfilled only once. Then, the *interestingness degree* of Ψ in t_4 is $\zeta(\Psi, t_4) = 0.5$ because it is activated twice, but fulfilled only once. Finally, in all the other traces t_2, t_3, t_5 and t_6 is $\zeta(\Psi, t) = 0$.

As we have just seen, the fulfilment of an RCon, in a trace, relies on the verification of the corresponding LTLp$_f$ formula over such a trace at the instant of activation. This process of verification of a formula φ on a trace can be achieved by constructing the related DFA $\mathcal{A}_\varphi$ and checking whether such trace is accepted by $\mathcal{A}_\varphi$ or not. To this extent, in the following, we give some other definitions and theorems.

Lemma 5.2. *Given a pure past formula $\varphi^{\blacktriangleleft}$, a pure present formula $\varphi^{\blacktriangledown}$, a pure future formula $\varphi^{\blacktriangleright}$, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, the following is holds true:*

- $t, i \models \varphi^{\blacktriangleleft} \equiv t_{[1,i]}, i \models \varphi^{\blacktriangleleft}$
- $t, i \models \varphi^{\blacktriangledown} \equiv t_{[i,i]}, i \models \varphi^{\blacktriangledown}$
- $t, i \models \varphi^{\blacktriangleright} \equiv t_{[i,n]}, i \models \varphi^{\blacktriangleright}$

The Lemma follows from the definition of the LTLp$_f$ semantics. It is trivial to see that having, at instant i, a pure past formula, its semantics only cares about events preceding i, whereas a pure future formula cares only about events following the instant i.

Theorem 5.3. *(Cecconi et al., 2018) Given an LTLp$_f$ formula φ, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, we have that $t, i \models \varphi$ iff $t_{[1,i]}, i \models \varphi^{\blacktriangleleft}, t_{[i,i]}, i \models \varphi^{\blacktriangledown}$ and $t_{[i,n]}, i \models \varphi^{\blacktriangleright}$ for at least a $(\varphi^{\blacktriangleleft}, \varphi^{\blacktriangledown}, \varphi^{\blacktriangleright}) \in \mathcal{S}(\varphi)$.*

The proof follows from Theorem 5.1 and Lemma 5.2.

Example 5.6. Let us consider the RCon $\Psi = a \mapsto (\ominus b \vee \Diamond c)$ with $\varphi = (\ominus b \vee \Diamond c)$, its separated formulas $\mathcal{S}(\varphi) = \{(\ominus b, True, True), (True, True, \Diamond c)\}$ and trace $t_1 = \langle d, f, a, f, c, a, f, b, a, f \rangle$ taken from Example 5.1.

- $t_1, 3 \models \varphi$ if, apart from the $True$ formulas that are satisfied, one of the following holds *true*:

  1. $\langle d, f, a \rangle, 3 \models \ominus b$
  2. $\langle a, f, c, a, f, b, a, f \rangle, 3 \models \Diamond c$

  since the latter holds *true*, φ is satisfied by $t_1(3)$.

- $t_1, 6 \models \varphi$ if, apart from the $True$ formulas that are satisfied, one of the following holds *true*:
  1. $\langle d, f, a, f, c, a \rangle, 6 \models \ominus b$
  2. $\langle a, f, b, a, f \rangle, 6 \models \Diamond c$

  since both are not satisfied, we can conclude that φ is not satisfied by $t_1(6)$.

- $t_1, 9 \models \varphi$ if, apart from the $True$ formulas that are satisfied, one of the following holds *true*:
  1. $\langle d, f, a, f, c, a, f, b, a \rangle, 9 \models \ominus b$
  2. $\langle a, f \rangle, 9 \models \Diamond c$

  since the former holds *true*, φ is satisfied by $t_1(9)$.

At this point, we can start talking about separated formulas verification on a trace using their corresponding DFAs.

Definition 5.8. Given a LTLp$_f$ formula φ, the *separated automata set* (sep.aut.set) $\mathcal{A}^{◀▼▶} \in 2^{\mathcal{A}\times\mathcal{A}\times\mathcal{A}}$ is the set of triples $\mathcal{A}^{◀▼▶} = (\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A} \times \mathcal{A} \times \mathcal{A}$ such that $\mathcal{A}^{◀} \doteq \mathcal{A}_{\varphi^{◀}}$, $\mathcal{A}^{▼} \doteq \mathcal{A}_{\varphi^{▼}}$ and $\mathcal{A}^{▶} \doteq \mathcal{A}_{\varphi^{▶}}$ for every $(\varphi^{◀}, \varphi^{▼}, \varphi^{▶}) \in \Psi$.

Here, we give the automata version of Example 5.4.

Example 5.7. Given the RCon $\Psi = a \mapsto (\ominus b \wedge True \wedge True) \vee (True \wedge True \wedge \Diamond c)$, its sep.aut.set is:

$$\mathcal{A}^{◀▼▶} = \{(\mathcal{A}_{\ominus b}, \mathcal{A}_{True}, \mathcal{A}_{True}), (\mathcal{A}_{True}, \mathcal{A}_{True}, \mathcal{A}_{\Diamond c})\}$$

Similarly to what we have seen before with Theorem 5.3, we can state the following:

Theorem 5.4. *(Cecconi et al., 2018) Given an* LTLp$_f$ *formula φ, its sep.aut.set $\mathcal{A}^{◀▼▶}$, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, we have that $t, i \models \varphi$ iff $t_{[1,i]}, i \in \mathcal{L}(\mathcal{A}^{◀})$, $t_{[i,i]}, i \in \mathcal{L}(\mathcal{A}^{▼})$ and $t_{[i,n]}, i \in \mathcal{L}(\mathcal{A}^{▶})$ for* at least *a $(\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A}^{◀▼▶}$.*

In the next Section, we will explain our modified approach which is a generalization of the Janus approach just described.

5.3 A generalization of the Janus Approach

The representation of an RCon (Definition 5.4), given in Cecconi et al. (2018), has two main drawbacks:

1. the activation condition α could be only a single task;
2. α is not incorporated in the formula φ,

where point 2 is a directed consequence of the RCon definition since the formula is verified only when the activation holds. In general, these two disadvantages restrict the Janus approach to what it can mine. In order to overcome this limitation, we have devised a generalization of the Janus approach, starting from the definition of RCon constraint.

Since, for Theorem 5.1, an LTLp$_f$ formula can always be rewritten as a disjunction of a conjuction of pure temporal formulas, hereafter we only deal with separated formulas. Hence, we define our RCon as follows:

Definition 5.9. Given an alphabet Σ, let $\bigvee_{j=1}^{m}(\varphi^{\blacktriangleleft} \wedge \varphi^{\blacktriangledown} \wedge \varphi^{\blacktriangleright})_j$ be a separated LTLp$_f$ formula over Σ. A Reactive Constraint (RCon) Ψ is defined as follows:

$$\Psi \doteq \bigvee_{j=1}^{m} (\varphi^{\blacktriangleleft} \wedge \varphi^{\blacktriangledown} \wedge \varphi^{\blacktriangleright})_j \tag{5.2}$$

We represent all the set of RCons over Σ as $\mathcal{R}$.

This new RCon is more general than the one given in Cecconi et al. (2018). The specification is given in the following terms:

- a formula on the past that checks if the past makes the triggering relevant;
- a propositional formula on the current instant that triggers potential interest on the instant itself;
- a formula on the future that must be satisfied for considering the current instant interesting.

Notice that to obtain the same representation of Cecconi et al. (2018), the present formula should be a conjunction between $\varphi^{\blacktriangledown}$ and the given activation condition α, merging α with separated formulas. Indeed, the activation condition could be triggered at each instant if it holds true in that instant.

Given the RCon expressed as in 5.2, we can define a new way to compute the *interestingness degree*.

Definition 5.10. Given a finite trace $t \in \Sigma$ of length n and an RCon $\Psi \doteq \bigvee_{j=1}^{m}(\varphi^{\blacktriangleleft} \wedge \varphi^{\blacktriangledown}_{(\wedge\alpha)} \wedge \varphi^{\blacktriangleright})_j$, we define the *interestingness degree* function $\eta : \mathcal{R} \times \Sigma^* \to [0,1] \subseteq \mathbb{R}$ as follows:

$$\eta(\Psi, t) = \begin{cases} \dfrac{|\{t \models \bigvee_{j=1}^{m}(\varphi^{\blacktriangleleft} \wedge \varphi^{\blacktriangledown} \wedge \varphi^{\blacktriangleright})_j\}|}{|\{t \models \bigvee_{j=1}^{m}\varphi^{\blacktriangledown}\}|}, & \text{if } |\{t \models \bigvee_{j=1}^{m}\varphi^{\blacktriangledown}\}| \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

Even the $\eta(\Psi, t)$ function measures how many times the RCon Ψ is interesting fulfilled with respect to the total number of activations within the trace t. Moreover, notice that with this representation the activation condition (embedded in the $\varphi^{\blacktriangledown}$) can also be a *propositional formula* rather than a single task. In order to show that the $\eta(\Psi, t)$ function is a generalization of the $\zeta(\Psi, t)$ function (Definition 5.7), given in Cecconi et al. (2018), we state the following Theorem.

Theorem 5.5. *The $\eta(\Psi, t)$ function is a generalization of the $\zeta(\Psi, t)$ function.*

Proof. In order to prove the Theorem, we show that from $\eta(\Psi, t)$ we derive $\zeta(\Psi, t)$. We focus only on the first part of the function because the other part is 0 for both functions. We introduce the $\hat{\eta}(\Psi, t)$ function as follows:

$$\hat{\eta}(\Psi, t) = \begin{cases} \dfrac{|\{t \models \bigvee_{j=1}^{m}(\varphi^{◀} \wedge (\varphi^{▼} \wedge \alpha) \wedge \varphi^{▶})_j \vee (False \wedge \alpha \wedge False)\}|}{|\{t \models \bigvee_{j=1}^{m}(\varphi^{▼} \wedge \alpha) \vee \alpha\}|}, & \text{if } |\{t \models \bigvee_{j=1}^{m}(\varphi^{▼} \wedge \alpha)\}| \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

As we can see, the $\hat{\eta}(\Psi, t)$ function is the $\eta(\Psi, t)$ function with the additional triple $(False \wedge \alpha \wedge False)$ at the numerator, α at the denominator and the conjunction $(\varphi^{▼} \wedge \alpha)$. These additions are needed to count all occurrences of activations. The former is given as a triple to be coherent with the notation. However, we can notice that the triple $(False \wedge \alpha \wedge False)$ evaluates always to false, so we can discard it. Similarly, at the denominator, we can discard $\bigvee_{j=1}^{m}(\varphi^{▼} \wedge \alpha)$ because of the presence of the added α. We thus obtain the following:

$$\hat{\eta}(\Psi, t) = \begin{cases} \dfrac{|\{t \models \bigvee_{j=1}^{m}(\varphi^{◀} \wedge (\varphi^{▼} \wedge \alpha) \wedge \varphi^{▶})_j\}|}{|\{t \models \alpha\}|}, & \text{if } |\{t \models \alpha\}| \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

Now, given that $\alpha \mapsto \varphi \equiv \bigvee_{j=1}^{m}(\varphi^{◀} \wedge (\varphi^{▼} \wedge \alpha) \wedge \varphi^{▶})_j$, $\hat{\eta}(\Psi, t)$ is equivalent to $\zeta(\Psi, t)$. $\square$

At this point, we can give all properties and consequences, introduced in Section 5.2, generalized thanks to Theorem 5.5. To begin with, the Lemma 5.2 remains unchanged. On the contrast, other theorems and definitions slightly change, since now we have a propositional formula on the current instant that triggers potential interest on the instant itself. In the following, we show their updated version.

Theorem 5.6. *Given an RCon $\Psi \doteq \bigvee_{j=1}^{m}(\varphi^{◀} \wedge \varphi^{▼} \wedge \varphi^{▶})_j$, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, we have that $t, i \models \Psi$ iff $t_{[1,i]}, i \models \varphi^{◀}, t_{[i,i]}, i \models \varphi^{▼}$ and $t_{[i,n]}, i \models \varphi^{▶}$ for at least a triple in Ψ.*

The proof follows from Lemma 5.2 and Theorem 5.5.

After that, regarding the evaluation of an RCon on the trace, we have what follows.

Definition 5.11. Given an RCon $\Psi \doteq \bigvee_{j=1}^{m}(\varphi^{◀} \wedge \varphi^{▼} \wedge \varphi^{▶})_j$, we define as *separated automata set* (sep.aut.set) $\mathcal{A}^{◀▼▶} \in 2^{\mathcal{A} \times \mathcal{A} \times \mathcal{A}}$ the set of triples $\mathcal{A}^{◀▼▶} = (\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A} \times \mathcal{A} \times \mathcal{A}$ such that $\mathcal{A}^{◀} \doteq \mathcal{A}_{\varphi^{◀}}$, $\mathcal{A}^{▼} \doteq \mathcal{A}_{\varphi^{▼}}$ and $\mathcal{A}^{▶} \doteq \mathcal{A}_{\varphi^{▶}}$ for every $(\varphi^{◀}, \varphi^{▼}, \varphi^{▶}) \in \Psi$.

Given the above-mentioned definition, we can generalize the Theorem 5.4 in:

Theorem 5.7. *Given an RCon $\Psi \doteq \bigvee_{j=1}^{m}(\varphi^{◀} \wedge \varphi^{▼} \wedge \varphi^{▶})_j$, its sep.aut.set $\mathcal{A}^{◀▼▶}$, a finite trace $t \in \Sigma^*$ of length n and an instant i, with $1 \leq i \leq n$, we have that $t, i \models \Psi$ iff $t_{[1,i]}, i \in \mathcal{L}(\mathcal{A}^{◀}), t_{[i,i]}, i \in \mathcal{L}(\mathcal{A}^{▼})$ and $t_{[i,n]}, i \in \mathcal{L}(\mathcal{A}^{▶})$ for at least a $(\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A}^{◀▼▶}$.*

To better capture the concept of sep.aut.set given before in Definition 5.11, we provide an example.

Example 5.8. Given the RCon $\Psi = (\ominus b \wedge a \wedge True) \vee (True \wedge a \wedge \Diamond c)$, its sep.aut.set is:

$$\mathcal{A}^{◀▼▶} = \{(\mathcal{A}_{\ominus b}, \mathcal{A}_a, \mathcal{A}_{True}), (\mathcal{A}_{True}, \mathcal{A}_a, \mathcal{A}_{\Diamond c})\}$$

In Section 5.4, we will see the implementation of the Janus algorithm for computing the *interestingness degree* of traces in real-world event logs, using RCons declared as in 5.2.

So far, we have described all theoretical results necessary for introducing and understanding how the Janus algorithm works. Now, we talk about automata generation given a pure past, pure present and a pure future formula possible thanks to our developed tool LTL$_f$2DFA.

Differently from what has been done in Cecconi et al. (2018) for the automata construction, in this thesis we propose a modified version of the Janus algorithm that works accordingly to our theoretical generalization and with LTL$_f$2DFA. Indeed, as already seen in Chapter 3, LTL$_f$2DFA is able to directly generate the minimum DFA for a pure past formula (PLTL) without passing through its pure future (LTL$_f$) reversed formula. In particular, LTL$_f$2DFA has been employed in the Janus algorithm for the generation of the automaton corresponding to each formula in the triple $(\varphi^{◀}, \varphi^{▼}, \varphi^{▶})$, for every triple belonging to Ψ. In Example 5.9, there are DFAs output from LTL$_f$2DFA.

Example 5.9. Let us consider the RCon $\Psi = (\ominus b \wedge a \wedge True) \vee (True \wedge a \wedge \Diamond c)$. The corresponding sep.aut.set $\mathcal{A}^{◀▼▶}$ for each triple in Ψ is depicted in Table 5.2. Moreover, notice that $\mathcal{A}^{▼}$ can be seen as a boolean flag. Indeed, it checks at each instant whether the activation condition holds true. In other words, if the activation condition is true at a given instant, it moves to the accepting state, otherwise it ends up to the error state.

5.3.1 Algorithm

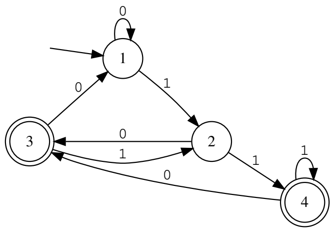
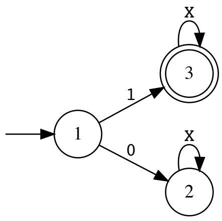
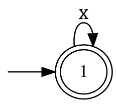
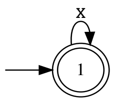
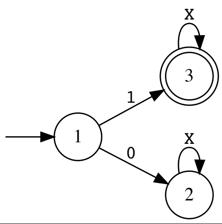
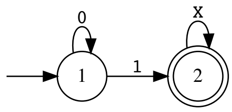
Here, we illustrate the modified version of the Janus algorithm present in (Cecconi et al., 2018). In particular, this version of the algorithm is able to deal with our new generalization of RCons introduced previously with Theorem 5.5 (Algorithm 5.1). The algorithm takes as input a trace and the sep.aut.set corresponding to an RCon expressed as 5.2.

How the algorithm works

As stated before, the goal of the algorithm is to compute the *interestingness degree* of an RCon with respect to a trace. The fundamental data structure is the *Janus state* ($\mathcal{J}$), a set of pairs consisting of a future automaton with its current state.

The naïve approach would count, whenever the activation condition holds, how many times there is *at least* a triple $(\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A}^{◀▼▶}$ that satisfies the RCon; then, it

Table 5.2. Representation of the separated automata set for $\Psi = a \mapsto (\ominus b \vee \Diamond c)$

$\mathcal{A}^{\blacktriangleleft} = \mathcal{A}_{\ominus b}$	$\mathcal{A}^{\blacktriangledown} = \mathcal{A}_{a}$	$\mathcal{A}^{\blacktriangleright} = \mathcal{A}_{true}$
		
$\mathcal{A}^{\blacktriangleleft} = \mathcal{A}_{true}$	$\mathcal{A}^{\blacktriangledown} = \mathcal{A}_{a}$	$\mathcal{A}^{\blacktriangleright} = \mathcal{A}_{\Diamond c}$
		

Algorithm 5.1. JANUS algorithm: given a a trace t, an RCon and its sep.aut.set $\mathcal{A}^{\blacktriangleleft\blacktriangledown\blacktriangleright}$, it returns the *interestingness degree*

$\mathcal{O} \leftarrow$ empty bag;
foreach event $t(i) \in t$ **do**
    foreach $(\mathcal{A}^{\blacktriangleleft}, \mathcal{A}^{\blacktriangledown}, \mathcal{A}^{\blacktriangleright}) \in \mathcal{A}^{\blacktriangleleft\blacktriangledown\blacktriangleright}$ **do**
        Make transition $t(i)$ on $\mathcal{A}^{\blacktriangleleft}$;
        if $\mathcal{A}^{\blacktriangledown}$ *accepts* $t(i)$ and $\mathcal{A}^{\blacktriangleleft}$ *is in an accepting state* **then** ▷ Activ. cond.
            $\mathcal{J} \leftarrow$ empty set of pairs;
            Add $(s_0^{\blacktriangleright}, \mathcal{A}^{\blacktriangleright})$ to $\mathcal{J}$;
            Add $\mathcal{J}$ to $\mathcal{O}$;
        end if
    end for
    foreach $\mathcal{J} \in \mathcal{O}$ **do**
        foreach $(s^{\blacktriangleright}, \mathcal{A}^{\blacktriangleright}) \in \mathcal{J}$ **do** $s^{\blacktriangleright} \leftarrow \delta^{\blacktriangleright}(s^{\blacktriangleright}, t(i))$ ▷ Make transition $t(i)$ on $\mathcal{A}^{\blacktriangleright}$
        end for
    end for
end for
if $\mid \mathcal{O} \mid > 0$ **then**
    return $\dfrac{\mid \{\mathcal{J} \in \mathcal{O} : \text{at least a } (s^{\blacktriangleright}, \mathcal{A}^{\blacktriangleright}) \in \mathcal{J}\} \text{ is such that } s^{\blacktriangleright} \in \mathcal{F}^{\blacktriangleright} \mid}{\mid \mathcal{O} \mid}$
else
    return 0
end if

would compute the ratio between this calculation with the overall activations. However, this would lead to an over and repeated computation. On the contrary, the Janus algorithm exploits the automata's property of remembering the past. In particular, for each event of the trace, the algorithm keeps updated the state of past automata and, when, at instant i, the activation condition holds, it relies on the fact that the past automata have already evaluated the trace until that instant (i.e. the sub-trace $t_{[1,i]}$). Then, for each triple in the sep.aut.set, only if the past automaton is in an accepting state and the present automaton accepts the current event $t(i)$, the algorithm initializes a *Janus state* with the future automaton with its current state. Each $\mathcal{J}$ is kept in memory by adding it to the bag $\mathcal{O}$. Subsequently, the algorithm updates only future automaton with their current state that appear in $\mathcal{O}$, namely, only relevant future automata are considered. As a result, an event in a trace interestingly fulfills an RCon if, on activation, *at least* a triple $(\mathcal{A}^{◀}, \mathcal{A}^{▼}, \mathcal{A}^{▶}) \in \mathcal{A}^{◀▼▶}$ is verified over the entire trace. By construction, this holds true if *at least* an automaton in $\mathcal{J}$ is, at the end, in a final state. Therefore, the *interestingness degree* is given by the ratio between such *Janus states* and the cardinality of $\mathcal{O}$.

Finally, unlike the Janus original version, our version has a different way of activation verification (line 5). While in the former version there is only a check of the form `event = activator`, in our version, since for Theorem 5.5 α is a propositional formula, we have to check if $\mathcal{A}^{▼}$ accepts the event.

5.4 Implementation

In this section, we fully describe the practical implementation of the Janus algorithm given in Section 5.3.1. In particular, we give some general information about its features, dependencies and usage. Then, we focus on the package explaining how is structured and commenting highlights on the code. The implementation on this thesis is called JANUS, is written in Python and is a porting of the Janus proof-of-concept software project written in Java.

To begin with, our implementation of the modified algorithm, presented in Section 5.3.1, together with the integration of the LTL$_f$2DFA tool, considerably differs from the original implementation. In particular, our JANUS can handle any type of constraint (not only DECLARE ones) and the user can specify the activation condition as a propositional formula instead of just one task. Nevertheless, the main goal of JANUS remains the same: compute the *interestingness degree* of traces on event log.

Secondly, JANUS provides I/O facilities for three different event log formats, namely simple *.txt* files, *.csv* files and *.xes* files for real-world event logs. Furthermore, the constraint is given as set of separated formulas as illustrated in 5.2.

JANUS requires Python>=3.6 and has the following dependencies:

- LTL$_f$2DFA, presented in Chapter 3. As stated before, it has been used for the generation of DFAS;
- OpyenXES, an open-source complete Python library for the XES Standard pub-

lished in (Valdivieso et al., 2018). It has been used for dealing with XES parsing and management.

The JANUS software is an open-source project and available to download on GitHub[2].

5.4.1 Package Structure

The structure of the JANUS package is relatively simple. It consists of the following:

- `janus.py`: it is the main module of the package. It contains the actual implementation of the Janus algorithm.
- `janus/`: it is the directory containing all the necessary code to correctly implement the algorithm. It has three subfolders:
  - `io/`: it contains the `InputHandler.py` which is in charge of handling the event log given as input.
  - `automata/`: it consists of the `automa.py` file, the `parserAutoma.py` file and the `sepautset.py` file. In this folder, we find all the code for dealing with automata.
  - `formulas/`: it comprises the `formula.py` file and the `separatedFormula.py`. These files defines the logic for LTLp$_f$ formulas and RCons.
- `files/`: this folder is the place where there are event logs. From this folder, a specific event log is parsed.

5.4.2 I/O

The `InputHandler.py` file, included in the `io/` folder, has been developed separately from the main module since we wanted to use our algorithm regardless of the input file format. In particular, thanks to the relative `InputHandler` class (Listing 5.1), the tool can import a log from a simple text file, from a csv and, finally, from a XES file. Hence, the JANUS tool can be used not only with the XES format, but also with other more manageable file formats.

Listing 5.1. The `InputHandler.py` module

```python
import csv
from opyenxes.data_in.XUniversalParser import XUniversalParser
from opyenxes.classification.XEventAttributeClassifier import \
XEventAttributeClassifier

class InputHandler:

    def __init__(self, input_path):
```

[2]https://github.com/Francesco17/janus

```
        self.input_path = input_path
        self._event_log = None
        self.load()

    @property
    def event_log(self):
        return self._event_log

    def load_txt(self):
        try:
            with open(self.input_path, 'r') as f:
                self._event_log = set(tuple(i) for i in \
                [f.read().splitlines()])
                f.close()
        except:
            raise IOError('[ERROR]: Unable to import text file')

    def load_csv(self):
        self._event_log = []
        try:
            with open(self.input_path, newline='', encoding='utf-8-sig') \
            as f:
                reader = csv.reader(f)
                for row in reader:
                    self._event_log.append(row[0])
        except:
            raise IOError('[ERROR]: Unable to import csv file')

    def load_xes(self):
        try:
            with open(self.input_path) as log_file:
                log = XUniversalParser().parse(log_file)[0]

            # get classifiers
            classifiers = []
            for cl in log.get_classifiers():
                classifiers.append(str(cl))

            classifier = XEventAttributeClassifier("activity", \
            [classifiers[0]])
            log_list = list(map(lambda trace: \
            (map(classifier.get_class_identity, trace)), log))

```

```
52              self._event_log = set(tuple(trace) for trace in log_list)
53
54          except:
55              raise IOError('[ERROR]:␣Unable␣to␣import␣xes␣file')
56
57      def load(self):
58          if self.input_path.endswith('.txt'):
59              self.load_txt()
60          elif self.input_path.endswith('.csv'):
61              self.load_csv()
62          elif self.input_path.endswith('.xes'):
63              self.load_xes()
64          else:
65              raise ValueError('[ERROR]:␣File␣extension␣not␣recognized')
```

From Listing 5.1, we can see that the `InputHandler` class has a main method called `load` that depending on the format of the file given as input calls the corresponding method specific for that format. If the format is not among *.txt*, *.csv* and *.xes*, it raises an error. Consequently, every specific method parses the event log. In particular, at line 37, the `load_xes` method takes advantage of the OpyenXES library APIs using its parser and classifier. In Section 5.4.5, we will look at how an `InputHandler` object can be instantiated.

5.4.3 Automata

In the `automata/` folder there are files devoted to handle and manage automata. Firstly, the `parserAutoma.py` module is a collection of functions used for parsing the *.dot* file and instantiating the data structure representing the automaton. In Listing 5.2 is shown that collection of functions.

Listing 5.2. The `parserAutoma.py` module

```
import pydot
from janus.automata.automa import Automa

def get_file(path):
    try:
        with open(path, 'r') as file:
            lines = file.readlines()
            file.close()
        return lines
    except IOError:
        print('[ERROR]:␣Not␣able␣to␣open␣the␣file␣from␣{}'.format(path))

def get_graph_from_dot(path):
```

```
    try:
        dot_graph = pydot.graph_from_dot_file(path)
        return dot_graph[0]
    except IOError:
        print('[ERROR]:␣Not␣able␣to␣import␣the␣dot␣file')

def get_final_label(label):

    s1 = label.replace("␣", "")
    s2 = s1.replace('"','')

    if s2 == '':
        return ['X']
    elif len(s2) < 2:
        return [s2]
    else:
        s3 = s2.replace(",","")
        s4 = s3.split('\\n')

        leng_elem = len(s4[0])
        temp = ''
        inter_label = []
        for i in range(leng_elem):
            for elem in s4:
                temp += elem[i]
            inter_label.append(temp)
            temp = ''

        return inter_label

def parse_dot(path, symbols):

    graph = get_graph_from_dot(path)

    nodes = []
    for node in graph.get_nodes():
        if node.get_name().isdigit():
            nodes.append(node.get_name())
        else: continue

    states = set(nodes)
    initial_state = sorted(nodes, key=int)[0]
```

```
57     lines = get_file(path)
58     accepting_states = set() # all accepting states of the automaton
59     for line in lines[7:]:
60         if line.strip() != 'node␣[shape␣=␣circle];':
61             temp = line.replace(";\n", "")
62             accepting_states.add(temp.strip())
63         else:
64             break
65 
66     sources = []
67     for elem in graph.get_edges():
68         if elem.get_source().isdigit():
69             sources.append(elem.get_source())
70         else: continue
71 
72     i = 0
73     transitions = dict()
74     for source in sources:
75         label = graph.get_edges()[i].get_label()
76         final_label = get_final_label(label)
77         destination = graph.get_edges()[i].get_destination()
78         i += 1
79         for lab in final_label:
80             if source in transitions:
81                 transitions[source][lab] = destination
82             else:
83                 transitions[source] = dict({lab: destination})
84 
85     #instantiation of automaton
86     automaton = Automa(
87         symbols=symbols,
88         alphabet={'0', '1', 'X'},
89         states=states,
90         initial_state=initial_state,
91         accepting_states=accepting_states,
92         transitions=transitions
93     )
94     return automaton
```

The most important function is called `parse_dot` (at line 44). It works as follows: given the path of a *.dot* file (the output of LTL$_f$2DFA) and symbols used in the formula, it returns an instantiation of the `Automa` class retrieving all information about the DFA, namely all its states, the initial state, accepting states and, finally, its transitions.

Afterwards, there is the `automa.py` in which the `Automa` class is implemented. This class is the data structure representing the DFA that is output from our tool LTL$_f$2DFA. It follows that in the `Automa` class there are methods able to perform transitions over the DFA, to tell whether if the automaton is in an accepting state or not and to tell whether an input symbols can be read by the automaton or not. In addition, when an object is instantiated, it is checked to be a valid DFA. In Listing 5.3 the `Automa` class implementation is shown.

Listing 5.3. The `automa.py` module

```python
import re

class Automa:
    """
        DFA Automa:
        - symbols          => list() ;
        - alphabet         => set() ;
        - states           => set() ;
        - initial_state    => str() ;
        - accepting_states => set() ;
        - transitions      => dict(), where
        **key**: *source* in states
        **value**: {*action*: *destination*}
    """

    def __init__(self, symbols, alphabet, states, initial_state, \
    accepting_states, transitions):
        self.symbols = symbols
        self.alphabet = alphabet
        self.states = states
        self._initial_state = initial_state
        self.accepting_states = accepting_states
        self.transitions = transitions
        self._current_state = self._initial_state
        self.validate()

    def valide_transition_start_states(self):
        for state in self.states:
            if state not in self.transitions:
                raise ValueError(
                    'transition start state {} is missing'.format(
                        state))

    def validate_initial_state(self):
        if self._initial_state not in self.states:
            raise ValueError('initial state is not defined as state')
```

```

    def validate_accepting_states(self):
        if any(not s in self.states for s in self.accepting_states):
            raise ValueError('accepting states not defined as state')

    def validate_input_symbols(self):
        alphabet_pattern = self.get_alphabet_pattern()
        for state in self.states:
            for action in self.transitions[state]:
                if not re.match(alphabet_pattern, action):
                    raise ValueError('invalid transition found')

    def get_alphabet_pattern(self):
        return re.compile("(^["+''.join(self.alphabet)+"]+$)")

    def validate(self):
        self.validate_initial_state()
        self.validate_accepting_states()
        self.valide_transition_start_states()
        self.validate_input_symbols()
        return True

    ...

    @property
    def current_state(self):
        return self._current_state

    @property
    def initial_state(self):
        return self._initial_state

    def check_others(self, action, map_symb_act, singleton):
        if singleton:
            del map_symb_act[action]
        else:
            for elem in action:
                del map_symb_act[elem]
        if all(value in {'0', 'X'} for value in map_symb_act.values()):
            return True
        else:
            return False

```

```
    def check_mixed_symbols(self, action):
        if len(action) == 1:
            return False
        else:
            for item in action:
                if item in self.symbols:
                    return True
                else: continue
            return False

    def make_transition(self, action):
        _action = None
        if len(action) == 1:
            _action = action[0]
        if len(self.symbols) == 0:
            self._current_state = \
            self.transitions[self._current_state]['X']

        for act in self.transitions[self.current_state].keys():
            temp = dict(zip(self.symbols, [value for value in act]))
            if set(action).issubset(set(self.symbols)):
                if len(self.symbols) == 1:
                    if temp[_action] in {'1','X'}:
                        self._current_state = \
                        self.transitions[self._current_state][act]
                    else: continue
                else:
                    if type(_action) == str:
                        if temp[_action] in {'1', 'X'} and \
                        self.check_others(_action, temp, True):
                            self._current_state = \
                            self.transitions[self._current_state][act]
                        else:
                            continue
                    else:
                        vals = [temp[_] for _ in action]
                        if set(vals).issubset({'1','X'}) and \
                        self.check_others(action, temp, False):
                            self._current_state = \
                            self.transitions[self._current_state][act]
                        else:
                            continue
            else:
```

```
123                 if self.check_mixed_symbols(action):
124                     common = [val for val in action if val in self.symbols]
125                     vals = [temp[_] for _ in common]
126                     if set(vals).issubset({'1', 'X'}) and \
127                     self.check_others(common, temp, False):
128                         self._current_state = \
129                         self.transitions[self._current_state][act]
130                     else:
131                         continue
132                 else:
133                     if all(value in {'0', 'X'} for value in temp.values()):
134                         self._current_state = \
135                         self.transitions[self._current_state][act]
136                     else:
137                         continue
138
139     def is_accepting(self):
140         if self._current_state in self.accepting_states:
141             return True
142         else:
143             return False
144
145     def accepts(self, input_symbol):
146         _current_state = self._current_state
147         self._current_state = self._initial_state
148         self.make_transition(input_symbol)
149         if self.is_accepting():
150             self._current_state = _current_state
151             return True
152         else:
153             self._current_state = _current_state
154             return False
```

Once an `Automa` object is instantiated, the method `validate` (line 52) checks whether the object is a valid DFA or not. In particular, it checks if the initial state and final states are actually states and it verifies that transitions are not made by invalid symbols. Then, the `make_transition` method at line 90 takes as input an action and make this action on the automaton, therefore modifying its current state. After that, the `is_accepting` method (line 139) simply tells whether the current state is accepting for the automaton itself or not. Finally, there is the `accepts` method at line 145, which given an input symbol returns *true* if it is accepted by the DFA.

As last module about automata, we illustrate the `sepautset.py`. It is the direct implementation of the *sep.aut.set* defined in 5.11. Indeed, this module contains the definition of the `SeparatedAutomataSet` class, namely the data structure that allow us to generate the corresponding *sep.aut.set*. Hence, it takes care of generating the set of separated automata starting from a list of separated formulas. We can see its implementation in Listing 5.4.

Listing 5.4. The `sepautset.py` module

```
from ltlf2dfa.Translator import Translator
from ltlf2dfa.DotHandler import DotHandler
from janus.automata.parserAutoma import parse_dot
import os, re

class SeparatedAutomataSet:

    def __init__(self, separated_formulas_set):
        self.separated_formulas_set = separated_formulas_set
        self._automa_set = self.compute_automa()

    @property
    def automa_set(self):
        return self._automa_set

    def build_automaton(self, triple):
        automata_list = []
        for formula in triple:
            symbols = re.findall('(?<![a-z])(?!true|false)[_a-z0-9]+', \
            str(formula))
            trans = Translator(formula)
            trans.formula_parser()
            trans.translate()
            trans.createMonafile(False) # true for DECLARE assumptions
            trans.invoke_mona() # returns inter-automa.dot
            dot = DotHandler()
            dot.modify_dot()
            dot.output_dot() # returns automa.dot
            automata_list.append(parse_dot("automa.dot", symbols))
            os.remove("automa.mona")
            os.remove("automa.dot")
            symbols = []
        return automata_list

    def compute_automa(self):
        result = []
        for triple in self.separated_formulas_set:
            past, present, future = self.build_automaton(triple)
            result.append( (past, present, future) )
```

```
40         return result
```

In this module, it has been employed the LTL$_f$2DFA tool. In fact, a `SeparatedAutomataSet` object receives a given set of separated formulas as input and it uses LTL$_f$2DFA to generate the DFA corresponding to each formula. Specifically, for each separated formula, i.e. a triple, (line 35) we compute the equivalent DFA on-line (line 16). This specific aspect represents a novelty with respect to what has been done in the original Java version of JANUS. So, unlike our JANUS version, the original one does not compute DFAs at execution time, but they are predefined (it only supports the most important DECLARE constraints) and given before the actual execution. Thus, this is a big step towards a complete generalization of the Janus algorithm implementation.

5.4.4 Formulas

Strictly connected to what we have just talked about in the previous section, in the `formulas/` folder there are modules in which we have defined the logic for managing and representing separated formulas and the formula constraint. In particular, we have the `separatedFormula.py` which comprises the `SeparatedFormula` class. Such class has the task of representing each triple resulting from the temporal separation (Definition 5.3). We show its implementation in Listing 5.5

Listing 5.5. The `separatedFormula.py` module

```
class SeparatedFormula:

    def __init__(self, triple):
        self.triple = triple
        self.validate()

    def validate(self):
        if len(self.triple) == 3:
            return True
        else:
            raise ValueError('[ERROR]:␣input␣is␣not␣a␣triple')

    def __str__(self):
        return str(self.triple)

    def __iter__(self):
        for elem in self.triple:
            yield elem
```

When instantiating a `SeparatedFormula` object, this is validated checking whether the given triple is valid.

After that, the other module contained in the `formulas/` folder is the `Formula.py` in which it is defined the `Formula` class. This class represents the formula of the RCon to be satisfied by traces. Actually, since the JANUS package works with already separated formulas, the `Formula` class gets a list of separated formulas, namely a list of triples. The implementation (Listing 5.6) of this class is quite similar to the `SeparatedFormula` class seen before.

Listing 5.6. The `Formula.py` module

```
from janus.formulas.separatedFormula import SeparatedFormula

class Formula:

    def __init__(self, separatedFormulas):
        self.separatedFormulas = separatedFormulas
        self.validate()

    def validate(self):
        if all(isinstance(x, SeparatedFormula) for x in \
        self.separatedFormulas) and self.separatedFormulas:
            return True
        else:
            raise ValueError('[ERROR]: Different types for conjuncts')

    def __str__(self):
        return ', '.join(map(str, self.separatedFormulas))

    def __iter__(self):
        for triple in self.separatedFormulas:
            yield triple
```

5.4.5 Main Module

Here, we describe the main module of the JANUS package. It is called `janus.py` and it contains the principal logic covering the Janus pseudocode anticipated in Section 5.3.1. We recall that in this version of JANUS we can specify any type of LTLp$_f$ formula as long as it is already separated following Definition 5.9. Moreover, our JANUS works with propositional formula as activation condition defined within the constraint formula. In Listing 5.7 is shown the code of the `janus.py` module.

Listing 5.7. The `janus.py` module

```
from janus.io.InputHandler import InputHandler
from janus.formulas.separatedFormula import SeparatedFormula
from janus.formulas.formula import Formula
from janus.automata.sepautset import SeparatedAutomataSet
import argparse, copy

```

```python
args_parser = argparse.ArgumentParser(description='Janus algorithm')
args_parser.add_argument('<event-log>', help='Path to the event log')

params = vars(args_parser.parse_args())

input_log = InputHandler(params['<event-log>'])
log_set = input_log.event_log

# separated formulas
sepFormula1 = SeparatedFormula(('Yerregistration', 'leucocytes', 'true'))
sepFormula2 = SeparatedFormula(('true', 'leucocytes', 'Fcrp'))

 # set manually the constraint
constraint = Formula([sepFormula1, sepFormula2])

sepautset = SeparatedAutomataSet(constraint).automa_set

for trace in log_set:
    print('[TRACE]: ' + str(trace))
    O = []
    for event in trace:
        for past, now, future in sepautset:
            past.make_transition([event.replace(' ','').lower()])
            if now.accepts([event.replace(' ','').lower()]) and
            past.is_accepting():
                J = {}
                temp = copy.deepcopy(future)
                J[temp] = future.initial_state
                O.append(J)
        for j in O:
            for aut, st in j.items():
                aut.make_transition([event.replace(' ','').lower()])
                j[aut] = aut.current_state

    if O:
        count = 0
        for janus in O:
            for automa, state in janus.items():
                if state in automa.accepting_states:
                    count += 1
                    break
                else:
                    continue
```

```
50          print('[ACTIVATED]:␣' + str(len(O)))
51          print('[FULFILLED]:␣' + str(count))
52          print('[INTERESTINGNESS␣DEGREE]:␣' + str(count/len(O)))
53      else:
54          print('[ACTIVATED]:␣0')
55          print('[FULFILLED]:␣0')
56          print('[INTERESTINGNESS␣DEGREE]:␣0')
```

As we can see, the practical implementation of the Janus algorithm directly reflects the pseudocode illustrated in 5.1.

Firstly, at line 12 we use the `InputHandler` class to parse and get the event log. Then, starting from line 16 we manually define our separated formulas composing the constraint. As a consequence, at lines 20 and 22, we define the constraint formula and the separated automata set, respectively. In particular, by defining the separated automata set we use our tool LTL$_f$2DFA for the automatic generation of automata. Finally, at this point (line 26), the computation of the *interestingness degree* can start.

5.5 Results

After having illustrated the whole implementation of JANUS, we are ready to present results of its execution where we have evaluated our tool against a real-world event log. Hence, we have analyzed the real-world event log called *Sepsis*[3], which reports trajectories of patients showing symptoms of sepsis in a Dutch hospital.

Since the *Sepsis* event log contains a lot of trajectories, we discuss, in the following example, the execution of JANUS only on a few of them.

Example 5.10. In this example, we show how JANUS computes the *interestingness degree* over different traces under the DECLARE assumption, namely when, at each instant, one and only one task is executed.

The constraint is expressed as in 5.2 and, for this example, is as follows:

$$\begin{aligned}\Psi = (&\ominus ER\ Registration \wedge Leucocytes \wedge True)\vee \\ &(True \wedge Leucocytes \wedge \Diamond CRP)\end{aligned}$$

In this example, we have selected the following three different traces:

1. {'ER Registration', 'IV Liquid', 'ER Triage', 'ER Sepsis Triage', 'LacticAcid', 'Leucocytes', 'CRP', 'IV Antibiotics', 'Admission NC', 'CRP', 'Leucocytes', 'Release A'};
2. {'ER Registration', 'ER Triage', 'ER Sepsis Triage', 'Admission NC', 'Release A'};
3. {'ER Registration', 'ER Triage', 'ER Sepsis Triage', 'Leucocytes', 'LacticAcid', 'CRP', 'IV Antibiotics', 'Admission NC', 'Leucocytes', 'CRP', 'CRP', 'Leucocytes', 'Release A'}.

[3] https://doi.org/10.4121/uuid:915d2bfb- 7e84- 49ad- a286- dc35f063a460

To begin with, in trace 1 the constraint is activated twice at instants $t_1(6)$ and $t_1(11)$, respectively. However, the constraint is satisfied only by the former activation because the triple $(True \wedge Leucocytes \wedge \Diamond CRP)$ is satisfied within the trace. Indeed, the second activation, at instant $t_1(11)$, does not fulfill the constraint since there is no triple of Ψ which holds true. So, the *interestingness degree* of this trace is $\frac{1}{2} = 0.5$.

Secondly, in trace 2, the constraint is never activated because the task "Leucocytes" never occurs, so no triple of Ψ can hold true. In this case, the *interestingness degree* is simply 0.

Lastly, in trace 3, the constraint is activated three times, respectively at instants $t_3(4)$, $t_3(9)$ and $t_3(12)$. Similarly to what happens in trace 1, here only the first two activations fulfill the constraint formula. In particular, also in this case, the triple of Ψ that is satisfied is $(True \wedge Leucocytes \wedge \Diamond CRP)$. As a result, the *interestingness degree* of this trace is $\frac{2}{3} = 0.667$.

Although Example 5.10 is a proof-of-concept of the theory given in previous sections, it does not show clearly the actual potential of the JANUS implementation given by the generalization Theorem 5.5 and the integration of our LTL$_f$2DFA, both introduced in this thesis. To this extent, we describe an *ad-hoc* example in which all contributions given in this thesis appear.

Example 5.11. In this example, we show how JANUS computes the *interestingness degree* over *ad-hoc* traces without having restrictions, namely:

- the constraint could be any kind of formula;
- we could have multiple symbols at each instant in the trace;
- the activation condition could be a propositional formula.

Therefore, the constraint is expressed as in 5.2 and, also for this example we borrow symbols from *Sepsis* as follows:

$$\Psi = (\ominus ER\ Registration \wedge (Leucocytes \wedge LacticAcid) \wedge True) \vee$$
$$(True \wedge (Leucocytes \wedge LacticAcid) \wedge \Diamond CRP)$$

As we can see, now we have chosen the activation condition as a propositional formula $(Leucocytes \wedge LacticAcid)$ meaning that the constraint will be activated only if, at a given instant of the trace, $(Leucocytes \wedge LacticAcid)$ holds true. Additionally, we have selected the following traces for the example:

1. {'ER Registration', ('ER Triage','ER Sepsis Triage), ('LacticAcid', 'IV Liquid'), ('Leucocytes', 'LacticAcid'), 'CRP', 'LacticAcid', ('Leucocytes', 'LacticAcid'), ('Leucocytes', 'IV Antibiotics'),'IV Liquid', 'Release A'}
2. {'ER Registration', ('ER Triage','ER Sepsis Triage), ('CRP', 'LacticAcid'), ('Leucocytes', 'LacticAcid'), 'Admission NC', 'CRP', 'LacticAcid', ('Leucocytes', 'IV Liquid'), ('Leucocytes', 'IV Antibiotics'), 'IV Liquid', 'Release A'}

Initially, in trace 1, the constraint is activated twice at instants $t_1(4)$ and $t_1(7)$, respectively. Yet, the constraint is satisfied only by the $t_1(4)$ where only the triple $(True \wedge (Leucocytes \wedge LacticAcid) \wedge \Diamond CRP)$ is satisfied within the trace. Notice that the constraint is activated only when in the trace there are at the same time *Leucocytes* and *LacticAcid*. Consequently, the *interestingness degree* of this trace is $\frac{1}{2} = 0.5$.

On the other hand, in trace 2, the constraint is activated and fulfilled once. Hence, in this case, the *interestingness degree* is simply 1.

5.6 Summary

In this chapter, we have presented how the LTL$_f$2DFA Python package has been well employed in the field of Business Process Management. In particular, we have explored the Janus approach to declarative process mining enhancing its peculiarities and, at the same time, giving our substantial contributions in generalizing the approach itself. After that, we have described the implementation of the janus algorithm, modified accordingly, highlighting all its main features. Finally, we have seen examples of execution results.

Chapter 6

Conclusions and Future Work

6.1 Overview

This thesis addressed the application of LTL and PLTL on finite traces in Planning and Declarative Process Mining. The variant of LTL evaluated on *finite* traces (LTL_f) and its past counterpart Past LTL (PLTL) have been extensively investigated in De Giacomo and Vardi (2013); Lichtenstein et al. (1985). In the CS literature, the modality to obtain a DFA from any LTL_f/PLTL formulas consisted of following the Algorithm 2.1 in Chapter 2. On the contrary, in this thesis, we followed the new approach analyzed in Zhu et al. (2017, 2018) to attain the corresponding DFA to a given LTL_f/PLTL formula. In particular, the translation procedure from an LTL_f/PLTL formula to its DFA entails the encoding of the LTL_f/PLTL formula to FOL and then feeding the MONA tool with this encoding. The execution of MONA with such an input yields the DFA as expected. Although the latter translation procedure is not straightforward, we have shown that, on a practical perspective, it performs relatively better.

This translation method has been applied both in Planning and Declarative Process Mining. In Planning scenario, while Camacho et al. (2017) analysed non-deterministic planning for LTL on finite and infinite traces goals, academics did not investigate planning for PLTL goals. We devised a solution to the problem of Planning for LTL_f/PLTL goals, that exploits the translation of temporal formulas to DFA, using LTL_f2DFA.

Regarding Declarative Process Mining, while Cecconi et al. (2018) introduced the Janus approach to compute the interestingness degree of traces in real event logs, by employing the augmentation of LTL_f with past modalities, we generalized this approach both theoretically and practically, employing the LTL_f2DFA tool.

6.2 Main Contributions

The main contributions of the thesis are multifold.

- We studied the theory behind the new approach, LTL_f2DFA, for translating an LTL_f/PLTL formula to the corresponding DFA, exploiting the power of the MONA tool. Then, we implemented the LTL$_f$2DFA tool. LTL$_f$2DFA is relevant in two regards. It is the first tool able to directly convert both LTL$_f$ and PLTL formulas to their corresponding DFA. Secondly, it adopts the MONA tool for the generation of automata. As a result, we contributed to advance researches in Zhu et al. (2017, 2018). In addition, the LTL$_f$2DFA is also available on Internet at the following website address: http://ltlf2dfa.diag.uniroma1.it.

- We explored how LTL$_f$ and PLTL can be used for expressing extended temporal goals in fully observable *non-deterministic* (FOND) planning problems. In these terms, we proposed a new approach, called FOND4LTL$_f$/PLTL, in compiling LTL$_f$/PLTL goals along with the original planning domain, specified in PDDL. The encoding of those temporal goals, directly in the PDDL domain and problem, relies on the result given by LTL$_f$2DFA. More precisely, our idea was the following: given a non-deterministic planning domain and an LTL$_f$/PLTL formula, we first obtained the corresponding DFA of the temporal formula through LTL$_f$2DFA, then, we encoded such a DFA into the non-deterministic planning domain. As a result, we reduced the original problem to a classic FOND planning problem. The absolute novelty is that, given the new LTL$_f$2DFA tool, it is now possible to express temporal extended goals not only in LTL$_f$, but also in PLTL (i.e. with past modalities), unlike former researches in this area of application (Camacho et al., 2017, 2018a,b). Furthermore, in order to integrate the FOND-SAT planner with FOND4LTL$_f$/PLTL, we updated the translation scripts of FOND-SAT. Finally, we developed a script able to parse the FOND-SAT result and build the transition system reporting all policies found.

- We gave an extension and generalization of the Janus algorithm in Cecconi et al. (2018). From a theoretical perspective, we introduced a new Theorem (Section 5.2) that proposes a new representation of the constraint formula allowing propositional formulas as activation condition, rather than a single task as in Cecconi et al. (2018). Moreover, from a practical perspective, we illustrated the Janus algorithm in Cecconi et al. (2018) modified accordingly with the new proposed theory. In these terms, we implemented this generalized approach by exploiting the power of the LTL$_f$2DFA tool. In such a scenario, LTL$_f$2DFA allowed to generate, at execution time, DFAs for any type of formula, overcoming the original limitation of the Janus approach to DECLARE constraints.

6.3 Future Works

The research work presented in this thesis seems to have raised more questions that it has answered. There are several lines of research arising from this work which should be pursued.

- We used both LTL$_f$ and PLTL, but always separately. We can also consider the LTLp$_f$ logic that merges the operators of the two. However, such a logic has not been studied yet from the computational point of view and while an approach based on translation to DFAs should be possible, the complexity of the translation is yet unknown.

- The LTL$_f$2DFA tool, introduced in Chapter 3, can be improved in several respects. Firstly, it is possible to improve the LTL$_f$-to-FOL translation procedure and to devise a better design and suited data structures for the problem. Secondly, the LTL$_f$2DFA tool should also be made more independent from hardware components, namely disk speed, as mentioned in Section 3.4.

- Theoretically, we proposed a new solution to the FOND planning for LTL$_f$/PLTL goals. However, it would be interesting to further our research to Partially Observable Non Deterministic (POND) planning for extended temporal goals. Next, the FOND4LTL$_f$/PLTL tool, shown in Chapter 4, can employ a multitude of planners rather than only FOND-SAT. In these terms, the user could choose among different planners and compare their solutions. Then, the FOND4LTL$_f$/PLTL tool can be extended to support full ADL. Finally, the performance of the FOND4LTL$_f$/PLTL tool should be compared to that of available competitors.

- Finally, regarding the JANUS approach, future researches could examine the possibility to directly provide the LTLp$_f$ formula rather than its separated formulas as input for the Janus algorithm. Alternatively, it is desirable to design and implement a tool able to automatically separate a given LTLp$_f$ formula. We acknowledge that our JANUS implementation could be improved in terms of performance and compare it to other declarative process mining techniques as done in Cecconi et al. (2018).

Bibliography

Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

Pascal Bercher. Pattern database heuristics for fully observable nondeterministic planning. 2010.

Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *International Workshop on Implementing Automata*, pages 6–25. Springer, 1996.

Alberto Camacho, Eleni Triantafillou, Christian J Muise, Jorge A Baier, and Sheila A McIlraith. Non-deterministic planning with temporally extended goals: Ltl over finite and infinite traces. In *AAAI*, pages 3716–3724, 2017.

Alberto Camacho, Jorge A Baier, Christian J Muise, and Sheila A McIlraith. Finite ltl synthesis as planning. In *ICAPS*, pages 29–38, 2018a.

Alberto Camacho, Christian J Muise, Jorge A Baier, and Sheila A McIlraith. Ltl realizability via safety and reachability games. In *IJCAI*, pages 4683–4691, 2018b.

Alessio Cecconi, Claudio Di Ciccio, Giuseppe De Giacomo, and Jan Mendling. Interestingness of traces in declarative process mining: The janus ltlpf approach. In *International Conference on Business Process Management*, pages 121–138. Springer, 2018.

AK Chandra, DC Kozen, LJ Stockmeyer, and J Alternation. Acm, vol. 28 (1), 1981.

Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147 (1-2):35–84, 2003.

Giuseppe De Giacomo and Sasha Rubin. Automata-theoretic foundations of fond planning for ltlf and ldlf goals. In *IJCAI*, pages 4729–4735, 2018.

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, volume 13, pages 854–860, 2013.

Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for ltl and ldl on finite traces. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1558–1564. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL `http://dl.acm.org/citation.cfm?id=2832415.2832466`.

Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. Reasoning on ltl on finite traces: Insensitivity to infiniteness. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 1027–1033. AAAI Press, 2014. URL `http://dl.acm.org/citation.cfm?id=2893873.2894033`.

Patrick Doherty and Jonas Kvarnstram. Talplanner: A temporal logic-based planner. *AI Magazine*, 22(3):95, 2001.

Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420. ACM, 1999.

Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, June/July 1998.

Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Vardi. *Reasoning about knowledge*. MIT press, 2004.

Jicheng Fu, Vincent Ng, Farokh B Bastani, I-Ling Yen, et al. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1949, 2011.

D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. Technical report, Jerusalem, Israel, Israel, 1997.

Dov Gabbay. The declarative past and imperative future. In *Temporal logic in specification*, pages 409–448. Springer, 1989.

Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1980.

Hector Geffner and Blai Bonet. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(1):1–141, 2013.

Tomas Geffner and Hector Geffner. Compact policies for fully-observable non-deterministic planning as sat. *arXiv preprint arXiv:1806.09455*, 2018.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

Valentin Goranko and Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2015 edition, 2015.

Steve Hanks and Drew V McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *AAAI*, volume 86, pages 328–333, 1986.

Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 89–110. Springer, 1995.

Hans Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California Los Angeles, 1968. URL `http://www.ims.uni-stuttgart.de/archiv/kamp/files/1968.kamp.thesis.pdf`. Published as Johan Anthony Willem Kamp.

Peter Kissmann and Stefan Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In *Annual Conference on Artificial Intelligence*, pages 1–8. Springer, 2009.

Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

Ugur Kuter, Dana Nau, Elnatan Reisner, and Robert P Goldman. Using classical planners to solve nondeterministic planning problems. In *Proceedings of the Eighteenth International Conference on International Conference on Automated Planning and Scheduling*, pages 190–197. AAAI Press, 2008.

Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In *Workshop on Logic of Programs*, pages 196–218. Springer, 1985.

Fabrizio M Maggi, RP Jagadeesh Chandra Bose, and Wil MP van der Aalst. A knowledge-based integrated approach for discovering and repairing declare maps. In *International Conference on Advanced Information Systems Engineering*, pages 433–448. Springer, 2013.

Nicolas Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79:122–128, 2003.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

Marco Montali. Declarative process mining. In *Specification and verification of declarative open interaction models*, pages 343–365. Springer, 2010.

Christian J Muise, Sheila A McIlraith, and J Christopher Beck. Improved non-deterministic planning by exploiting state relevance. In *ICAPS*, 2012.

M. Pesic. *Constraint-based workflow management systems: shifting control to users.* PhD thesis, Department of Industrial Engineering & Innovation Sciences, 2008.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL `https://doi.org/10.1109/SFCS.1977.32`.

M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959. ISSN 0018-8646. doi: 10.1147/rd.32.0114. URL `http://dx.doi.org/10.1147/rd.32.0114`.

Miquel Ramirez and Sebastian Sardina. Directed fixed-point regression-based planning for non-deterministic domains. In *ICAPS*, 2014.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3837. URL `http://doi.acm.org/10.1145/3828.3837`.

Hernan Valdivieso, Wai Lam Jonathan Lee, Jorge Munoz-Gama, and Marcos Sepúlveda. Opyenxes: A complete python library for the extensible event stream standard. In *Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018.*, pages 71–75, 2018. URL `http://ceur-ws.org/Vol-2196/BPM_2018_paper_15.pdf`.

Shufang Zhu, Lucas M Tabajara, Jianwen Li, Geguang Pu, and Moshe Y Vardi. Symbolic ltlf synthesis. *arXiv preprint arXiv:1705.08426*, 2017.

Shufang Zhu, Pu Geguang, and Moshe Y. Vardi. First-order vs. second-order for ltlf-to-automata: An extended abstract. Women in Logic, 2018.